Festschrift

LNCS 5765

Gregor Engels Claus Lewerentz
Wilhelm Schäfer Andy Schürr
Bernhard Westfechtel (Eds.)

Graph Transformations and Model-Driven Engineering

Essays Dedicated to Manfred Nagl on the Occasion of His 65th Birthday

Springer

Lecture Notes in Computer Science 5765

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Gregor Engels Claus Lewerentz
Wilhelm Schäfer Andy Schürr
Bernhard Westfechtel (Eds.)

Graph Transformations and Model-Driven Engineering

Essays Dedicated to Manfred Nagl
on the Occasion of his 65th Birthday

Springer

Volume Editors

Gregor Engels
University of Paderborn
33098 Paderborn, Germany
E-mail: engels@uni-paderborn.de

Claus Lewerentz
Brandenburg Technical University at Cottbus
03046 Cottbus, Germany
E-mail: cl@tu-cottbus.de

Wilhelm Schäfer
University of Paderborn
33098 Paderborn, Germany
E-mail: wilhelm@upb.de

Andy Schürr
Technische Universität Darmstadt
64283 Darmstadt, Germany
E-mail: andy.schuerr@es.tu-darmstadt.de

Bernhard Westfechtel
University of Bayreuth
95447 Bayreuth, Germany
E-mail: Bernhard.Westfechtel@uni-bayreuth.de

Cover illustration:
The illustration appearing on the cover of this book is the work of Anna Tait.

Library of Congress Control Number: 2010939155

CR Subject Classification (1998): D.2, F.3, D.3, D.2.4, C.2, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-17321-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-17321-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Manfred Nagl

Preface

Manfred Nagl has been a very active, productive researcher with great impact in a number of different areas, e.g., graph transformations and their applications to a wide range of disciplines, software engineering environments, engineering design processes, and software architectures. We — five of his numerous academic descendants — were influenced deeply by Manfred's work. For this reason, we decided to prepare this volume, which was edited in his honor on the occasion of his 65th birthday. A "pre-release" (book of abstracts) was presented to Manfred at a celebration at RWTH Aachen University in June 2009. The complete volume followed when Manfred received an honorary doctorate from the University of Paderborn in November 2010.

Altogether, we collected 30 papers. The types of papers vary significantly, including classic research papers in the style of journal articles, surveys of focused research areas, essays reflecting on certain research topics, and papers summarizing long-term work conducted by Manfred Nagl.

All papers were subject to a thorough quality control process involving at least two reviews for each paper. The editors were assisted by numerous additional reviewers, whose work is gratefully acknowledged.

The volume is structured into five parts, each of which was managed by one of the editors:

- Graph Transformations — Theory and Applications (Andy Schürr, Darmstadt University of Technology)
- Software Architectures and Reengineering (Claus Lewerentz, Brandenburg University of Technology)
- Process Support (Gregor Engels, University of Paderborn)
- Embedded Systems Engineering (Wilhelm Schäfer, University of Paderborn)
- Engineering Design Applications (Bernhard Westfechtel, University of Bayreuth)

We would like to thank all authors for contributing to the high quality of this volume. Special thanks to Bernhard Rumpe, who inspired and motivated us to edit a Festschrift in honor of Manfred Nagl.

November 2010

Gregor Engels
Claus Lewerentz
Wilhelm Schäfer
Andy Schürr
Bernhard Westfechtel

Organization

Editors

Gregor Engels	University of Paderborn
Claus Lewerentz	Brandenburg University of Technology, Cottbus
Wilhelm Schäfer	University of Paderborn
Andy Schürr	Darmstadt University of Technology
Bernhard Westfechtel	University of Bayreuth

Additional Reviewers

Uwe Assmann	Reiko Heckel	Klaus Pohl
Simon Becker	Dominik Henrich	Arend Rensink
Dirk Beyer	Stefan Jablonski	Kurt Schneider
Andrea Corradini	Matthias Jarke	Oliver Sudmann
Jürgen Ebert	Gabor Karsai	Gabriele Taentzer
Tobias Eckardt	Hans-Jörg Kreowski	Josée Tassé
Hartmut Ehrig	Sabine Kuske	Daniel Varro
Wolfgang Emmerich	Juan de Lara	Heike Wehrheim
Carlo Ghezzi	Renate Löffler	Andreas Wiesner
Holger Giese	Jan Meyer	Albert Zündorf
Martin Glinz	Mark Minas	
Joel Greenyer	Chris Paredis	

Table of Contents

Graph Transformations: Theory and Applications

Software Architectures and Reengineering

Process Support

Embedded Systems Engineering

Engineering Design Applications

Graph Transformations and Model-Driven Engineering: The Merits of Manfred Nagl

Gregor Engels[1], Claus Lewerentz[2], Wilhelm Schäfer[3], Andy Schürr[4], and Bernhard Westfechtel[5]

[1] Database and Information Systems, University of Paderborn, Paderborn, Germany
[2] Software Systems Engineering, Brandenburg University of Technology, Cottbus, Germany
[3] Software Engineering, University of Paderborn, Paderborn, Germany
[4] Real-Time Systems Lab, Darmstadt University of Technology, Darmstadt, Germany
[5] Applied Computer Science I (Software Engineering), University of Bayreuth, Bayreuth, Germany

1 Short CV

Manfred was born in 1944 in Landskron, Czechia. In 1963, he enrolled for mathematics and physics at the University of Erlangen-Nuremberg, from which he received a master degree in 1969. Subsequently, he worked at a research lab of Siemens AG in the area of graphics software. In 1971, he returned to the University of Erlangen-Nuremberg. At the chair of programming languages headed by Hans-Jürgen Schneider, he focused on graph grammars and graph rewriting systems. Manfred received a doctoral degree in 1974 and a habilitation degree in 1979, both from the Engineering Faculty of the University of Erlangen-Nuremberg.

In 1979, Manfred obtained his first position as a professor at the University of Koblenz-Landau, where he worked as an associate professor of computer science until 1981. He then moved to the University of Osnabrück, where he held the chair of applied computer science as a full professor until 1986. From then on, Manfred held a chair of computer science at RWTH Aachen University until he retired in July 2009.

2 Research

In 1971, Manfred started working on *graph transformations* (graph grammars and graph rewriting systems), which was a very young field at that time. In fact, the first publications on graph grammars (by Manfred's advisor, Hans-Jürgen Schneider, and John Pfaltz) appeared around 1970. The first years of research were dominated by work on theoretical foundations. Manfred contributed to the theory of graph transformations by developing the set-theoretic approach, which significantly differs from the categorical approach of Hans-Jürgen Schneider. Furthermore, Manfred put strong emphasis on implementations and applications, as it is documented by his habilitation thesis "Graph Grammars: Theory, Applications, Implementation", which was published as a text book in 1979.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 1–5, 2010.

Graph transformations constitute the most important thread of research which Manfred has been constantly developing up to the present. Around 1980, the field of *integrated software engineering environments* emerged. While programs were developed traditionally with separate tools such as text editors, compilers, and debuggers, an integrated software engineering environment was envisioned consisting of tightly integrated tools operating on a common program representation. While abstract syntax trees and attribute grammars were employed in most other projects, Manfred favored graphs for the internal program representation and graph transformation systems for specifying operations on program graphs. Manfred launched the *IPSEN* project, where pioneering work on implementing and applying graphs and graph transformations in integrated software engineering environments was accomplished. The IPSEN project was started at the University of Osnabrück in 1982 and was continued at RWTH Aachen University until 1996, when it was finally documented in an LNCS volume (LNCS 1170).

The IPSEN environment is a complex software system, the development of which required a thoroughly devised and well-structured *software architecture*. This research field emerged as a hot topic only around 1995, although Manfred started working on software architecture much earlier (around 1980). Several doctoral dissertations performed under his supervision were devoted to software architecture. In 1990, Manfred published — to the best of our knowledge — the first text book on software architecture.

Conceptually, the IPSEN environment was based on graph transformations. However, graph transformation systems had to be written by hand and had to be converted manually into an implementation on top of a graph-based database management system. At the end of the 1980s, the development of *PROGRES* was started. PROGRES is both a language for specifying graph transformation systems and an integrated development environment, which was built with the help of IPSEN. Phrased in current terms, PROGRES supports *model-driven engineering* with graph transformations. Again, Manfred initiated pioneering work in an area which became a hot topic much later. A fully functional version of PROGRES was available around 1996, and PROGRES was used for many research projects under the supervision of Manfred and his descendants until his retirement. Nowadays, numerous languages and tools for model-driven engineering are available, partly inspired by work on graph transformations, and are starting to impact industrial software development.

RWTH Aachen is a university with strong emphasis on engineering disciplines. In 1990, Manfred launched a joint project with mechanical engineers which was concerned with *process support* for *engineering design*. From the very beginning, Manfred had a unified view on design processes in engineering disciplines and software processes. In 1997, he managed to set up a large research project — the Collaborative Research Council *IMPROVE* — which was dedicated to models and tools for improving design processes in chemical engineering. The results of this project were documented in an LNCS volume (LNCS 4970), which appeared in 2008. Within the IMPROVE project, a series of research prototypes

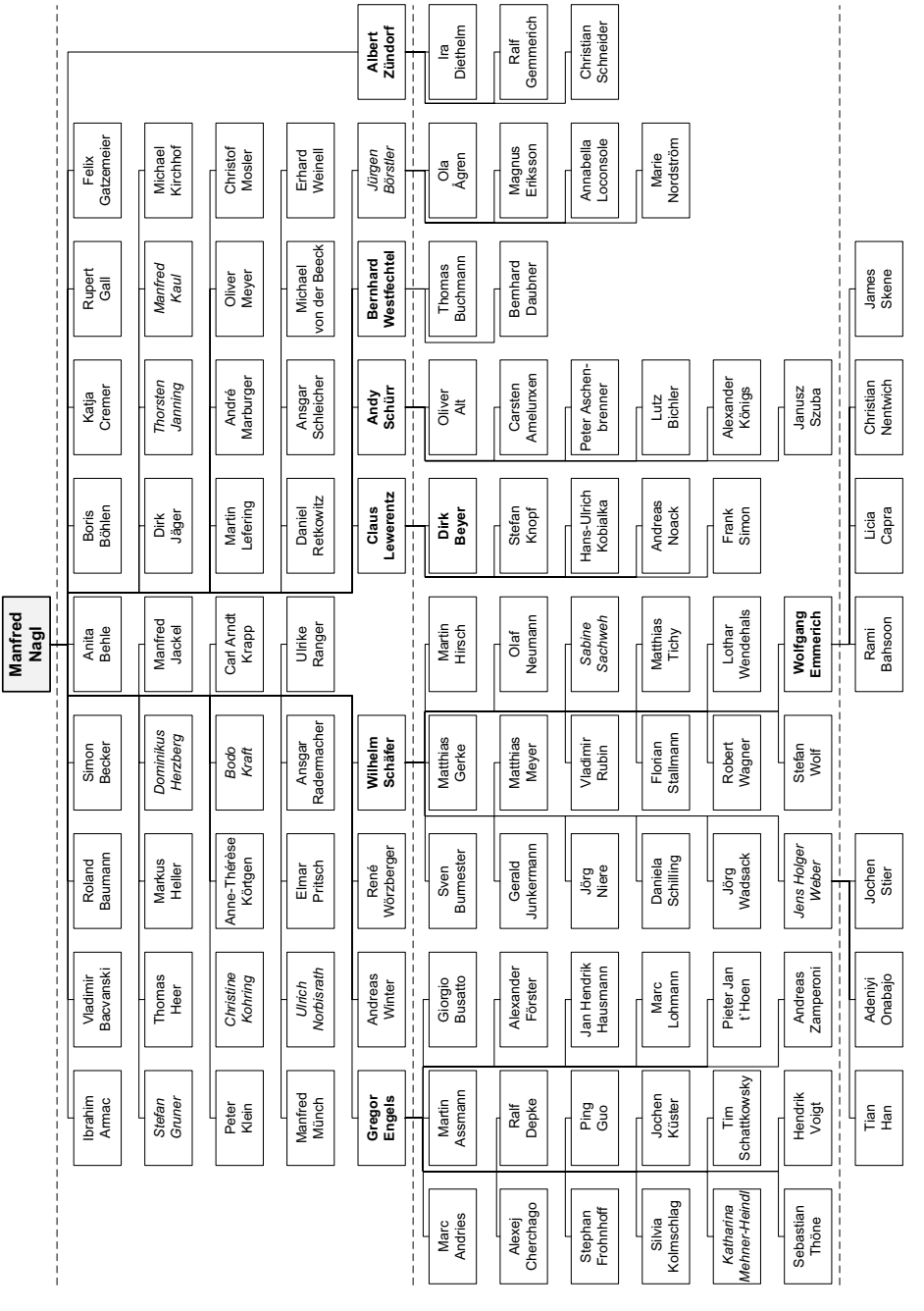


Fig. 1. Descendants of Manfred Nagl

of integrated process support environments was developed. These environments were partly based on the graph transformation technology described above.

3 Students

Manfred has created a large number of scientific descendants, who are listed in Figure 1 (without any guarantee for completeness). Up to now, 45 Ph.D. students obtained their doctoral degrees; they constitute the first-level branches of the "family tree". Among them, 6 researchers currently hold full professorship positions in software engineering at universities in Germany (shown in bold face). In addition, there are 8 descendants holding positions as associate professors, professors at universities of applied sciences, or assistant professors (emphasized in italics). The figure also shows 56 "grand children" (among them 5 professors) and even 7 "great-grand children". Altogether, the "family tree" provides an impressive demonstration of Manfred's impact.

4 Publications

Manfred has produced an impressive list of publications, which currently comprises 24 authored and edited books and 126 refereed articles for journals, conferences, and workshops. Below, only the books are listed. Among them, we would like to emphasize

- the first text book on graph transformations [3],
- the first German text book on Ada [4],
- the first text book on software architecture [10],
- the LNCS volume on the IPSEN project [13],
- the LNCS volume on the IMPROVE project [22], and
- proceedings of various workshops and conferences related to graphs and graph transformations [2,5,6,7,9,12,17,20,21,23].

References

1. Schneider, H.J., Nagl, M. (eds.): Programmiersprachen, 4. Fachtagung der Gesellschaft für Informatik. Informatik-Fachberichte 1. Springer, Berlin (1976)
2. Nagl, M., Schneider, H.J. (eds.): Graphs, Data Structures, Algorithms, Proceedings Workshop on Graph-Theoretic Concepts in Computer Science (WG 1978). Applied Computer Science, vol. 13. Carl Hanser Verlag, München (1979)
3. Nagl, M.: Graph-Grammatiken: Theorie, Anwendungen, Implementierung. Vieweg, Braunschweig/Wiesbaden (1979)
4. Nagl, M.: Einführung in die Programmiersprache Ada. Vieweg, Braunschweig/Wiesbaden (1982)
5. Ehrig, H., Nagl, M., Rozenberg, G. (eds.): Graph Grammars and Their Application to Computer Science. LNCS, vol. 153. Springer, Heidelberg (1983)
6. Nagl, M., Perl, J. (eds.): Proceedings International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1983). Trauner-Verlag, Linz (1984)

7. Ehrig, H., Nagl, M., Rozenberg, G. (eds.): Graph Grammars and Their Application to Computer Science. LNCS, vol. 291. Springer, Heidelberg (1987)
8. Nagl, M.: Einführung in die Programmiersprache Ada, 2. new and extended edn. Vieweg, Wiesbaden (1988); and two further editions in 1991 and 1992
9. Nagl, M. (ed.): Graph-Theoretic Concepts in Computer Science (WG 1989). LNCS, vol. 411. Springer, Heidelberg (1990)
10. Nagl, M.: Softwaretechnik: Methodisches Programmieren im Großen. Springer, Heidelberg (1990)
11. Nagl, M. (ed.): Software- und Information Engineering, Congress VI, Online-Konferenz 1993. Online-Verlag, Velbert (1993)
12. Nagl, M. (ed.): Graph-Theoretic Concepts in Computer Science (WG 1995). LNCS, vol. 1017. Springer, Heidelberg (1995)
13. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996)
14. Nagl, M. (ed.): Verteilte, integrierte Anwendungsarchitekturen: Die Software-Welt im Umbruch, Congressband VI, Online 1997. Online-Verlag, Velbert (1997)
15. Nagl, M., Westfechtel, B. (eds.): Integration von Entwicklungssystemen in Ingenieuranwendungen — Substantielle Verbesserung der Entwicklungsprozesse. Springer, Berlin (1999)
16. Nagl, M.: Softwaretechnik und Ada 1995 — Entwicklung großer Systeme, 5. new and extended edition. Vieweg, Wiesbaden (1999), 6. edn. (2003)
17. Nagl, M., Schürr, A., Münch, M. (eds.): Applications of Graph Transformations with Industrial Relevance: First International Workshop (AGTIVE 1999). LNCS, vol. 1779. Springer, Heidelberg (2000)
18. Nagl, M. (ed.): B2B mit EAI: Strategien mit XML, Java + Agenten, Online 2001, Congressband VI. Online-Verlag, Velbert (2001)
19. Nagl, M., Westfechtel, B. (eds.): Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen. John Wiley VCH Verlag, Weinheim (2003)
20. Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.): Applications of Graph Transformations with Industrial Relevance: Second International Workshop (AGTIVE 2003). LNCS, vol. 3062. Springer, Heidelberg (2004)
21. Hromkovič, J., Nagl, M., Westfechtel, B. (eds.): Graph-Theoretic Concepts in Computer Science: 30th International Workshop (WG 2004). LNCS, vol. 3353. Springer, Heidelberg (2004)
22. Nagl, M., Marquardt, W. (eds.): Collaborative and Distributed Chemical Engineering: From Understanding to Substantial Design Process Support — Results of the IMPROVE Project. LNCS, vol. 4970. Springer, Heidelberg (2008)
23. Schürr, A., Nagl, M., Zündorf, A. (eds.): Applications of Graph Transformations with Industrial Relevance: Third International Symposium (AGTIVE 2007). LNCS, vol. 5088. Springer, Heidelberg (2008)
24. Nagl, M., Bargstädt, H., Hoffmann, M., Müller, N.: Zukunft Ingenieurwissenschaften - Zukunft Deutschland, Beiträge einer 4ING-Fachkonferenz und der gemeinsamen Plenarversammlung der 4ING-Fakultätentage. Springer, Berlin (2008)

The Edge of Graph Transformation — Graphs for Behavioural Specification

Arend Rensink

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. The title of this paper, besides being a pun, can be taken to mean either the *frontier of research* in graph transformation, or the *advantage of using* graph transformation. To focus on the latter: Why should anyone not already educated in the field adopt graph transformation-based methods, rather than a mainstream modelling language or a process algebra; or vice versa, what is holding potential users back? These questions can be further refined by focusing on particular aspects like usability (available tools) or power (available theory).

In this paper, we take a fresh and honest look at these issues. Our perspective is the use of graph transformation as a formalism for the specification and analysis of system behaviour. There is no question that the general nature of graphs is at once their prime selling point (essentially everything can be specified in terms of graphs) and their main drawback (the manipulation of graphs is complex, and many properties that are useful in more specialised formalisms no longer hold for general graphs).

The outcome of this paper is a series of recommendations that can be used to outline a research and development programme for the coming decade. This may help to stimulate the continued and increasing acceptance of graph transformation within the rest of the scientific community, thereby ensuring research that is relevant, innovative and on the edge.

1 Background

In this paper we take a look at the advantages and disadvantages of graph transformation as an underlying formalism for the specification and analysis of system behaviour. For this purpose we review criteria for such a basic formalism, and we discuss how well graph transformation meets them in comparison with other formalisms.

We will start with a couple of observations. First of all, the field of graph transformation is quite broad: there are many formalisms that differ in philosophy as well as technical details, and yet all fall under the header of graph transformation — for an overview see the series of handbooks [65,23,26]. One often used classification distinguishes the algorithmic approach, pioneered by Nagl [53] among others, from the algebraic approach, pioneered by Ehrig among others [27]. We do not claim to be comprehensive in this paper; although we will not always state so explicitly, in our own research we are leaning towards the algebraic interpretation as the one closest to other, non-graph-based formalisms for behavioural specification, such as process algebra and Petri nets. Our findings are undoubtedly coloured by this bias.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 6–32, 2010.

The next observation is that the phrase "graph transformation for the specification of system behaviour" is up for more than one interpretation.

Graph transformation as a modelling language. In this approach, system states are directly captured through graphs, and system behaviour through graph transformation rules. Thus, graph transformation is on the interface with the user (designer/programmer). This is the approach assumed by many graph transformation tools: prime examples are FUJABA [30], which has an elaborate graphical front-end for this purpose, and AUGUR2 [44], which requires rather specialised graph transformation rules as input.

Graph transformation for operational semantics. In this interpretation, system behaviour is captured using another modelling language, for which there exists a graph transformation-based semantics. Thus, graph transformation is hiding "under the hood" of the other modelling language. There has been quite some research in the definition of such operational semantics: for instance, for statecharts [45], activity diagrams [35,28] and sequence diagrams [36,11] and for object-oriented languages [12,41,64].

Note that this distinction can also be made outside the context of behavioural specification: for instance in model transformation, where again there are proposals to use graph transformation as a modelling language, for instance in the form of Triple Graph Grammars [70], or to define a semantics for QVT [47].

Both scenarios have their attractions; however, the criteria for the suitability of the underlying formalism do depend on which of them is adhered to. For instance, in the first scenario (graph transformation as a modelling language), the visual nature of graphs is a big advantage: in fact, many articles promoting the use of graph transformation stress this point. On the other hand, this criterion is not at all relevant in the second scenario (graph transformation for operational semantics) — at least not to the average user of the actual modelling language, though it may still be important to parties that have to draw up, understand or use the operational semantics. Instead, in this second scenario, graph transformation is competing with formalisms with far less intuitive appeal but more efficient algorithms, for instance based on trees (as in ordinary Structural Operational Semantics) or other data structures. In this paper we consider both scenarios.

In the next section, we first discuss some of the existing variations on graphs, emphasising their relative advantages. In Section 3 we proceed to step through a number of criteria that are (to a varying degree) relevant in a formalism for behavioural specification and analysis, and we evaluate how well graph transformation does in meeting these criteria. In Section 4, we evaluate the results and come to some recommendations on topics for future research.

Disclaimer. It should perhaps be stressed that this paper represents a personal view, based on past experience of the author, not all of which is well documented. Where possible we will provide evidence for our opinions, but elsewhere we do not hesitate to rely on our subjective impressions and intuitions.

2 A Roadmap through the Graph Zoo

Whatever the precise scenario in which one plans to use graph transformation, the first decision to take is the actual notion of graphs to be used. There are many possible choices, and they can have appreciable impact. In this section we discuss some of the dimensions of choice. In this, our perspective is coloured in favour of the so-called algebraic approach. A related discussion, which distinguishes between *glueing* and *connecting* approaches, can be found in [6].

We limit ourselves to directed graphs, as these are ubiquitous in the realm of graph transformation. In addition, essentially all our graphs are edge-labelled, though in some cases the labels are indirectly assigned through typing. True node labels will only appear in the context of typing, though a poor man's substitute can be obtained by employing special edges.[1] We will not bother about the structure of labels: instead, a single set Lab will serve as a universe of labels for all our graphs.

We will refrain from formally defining graph morphisms for all the notions of graph under review, as the purpose of this section is not to give an exhaustive formal overview but rather to convey intuitions.

2.1 Nodes for Entities, Edges for Relations

The natural interpretation of a graph, when drawn in a figure, is that the nodes represent entities or concepts in the problem domain. Thus, each node has an identity that has a meaning to the reader. In the context of the figure, the identities are determined by the two-dimensional positions of the nodes, but obviously these are not the same as the identities in the reader's mind: instead, the reader mentally establishes a mapping from the nodes to the entities in his interpretation. Typically, though not universally, this mental mapping is an injection; or, if we think of the mental model as encompassing only those entities that are depicted in the graph, it is a bijection.

The edges play quite a different role in such a natural interpretation. Edges serve to connect nodes in particular ways: every edge stands for a relation between its end nodes, where the edge label tells what kind of relation it is. Thus, one particular difference with nodes is that edges do not themselves have an identity. This is reflected by the fact that, in this interpretation, it does not make sense for two nodes to be related "more than once" in the same way: in other words, there cannot be *parallel edges*, with the same end nodes and the same label. One may also think of this as a *logical* interpretation, in the sense that the collection of edges with a given label is analogous to a binary predicate over the nodes.

This "natural" or "logical" interpretation is captured by the following definition.

Definition 1 (simple graph). *A simple graph is a tuple $\langle V, E \rangle$, where V is an arbitrary set of nodes and $E \subseteq V \times \mathsf{Lab} \times V$ is a set of edges. There are derived functions $src, tgt\colon E \to V$ and $lab\colon E \to \mathsf{Lab}$ projecting each edge onto, respectively, its first, third and second component.*

[1] Note that this is not necessarily true outside our context of using graphs for behavioural modelling: for instance, in the field of graph grammars, explicit node labels may be necessary to distinguish non-terminals.

Choosing this notion of graphs has important consequences for the corresponding notion of transformation.

- *Simple graphs do not fit smoothly into the algebraic framework.* (This is not really surprising, as simple graphs are relational models rather than algebraic ones.) Unless one severely restricts the addition and deletion of edges (namely, to rules where an edge is only added or deleted if one of its end nodes is at the same time also added or deleted), the theory of *adhesive categories* that is the flagship of algebraic graph transformation does not apply. Instead, algebraically the transformation of simple graphs is captured better by so-called *single pushout rewriting*, which is far less rich in results. See Section 2.7 below for a more extensive discussion on these issues.
- *Adding a relation may fail to change a graph.* One of the basic operations of a graph transformation rule is to add an edge between existing nodes. In simple graphs, if an edge with that label already exists between those nodes, the rule is applicable but the graph is unchanged. This is in some cases counter-intuitive. It may be avoided by including a test for the absence of such an edge.
- *Node deletion removes all incident edges.* In the natural notion of simple graph (single-pushout) rewriting mentioned above, nodes can always be deleted by a rule, even if they have incoming or outgoing edges that the rule does not explicitly delete as well. From the edges-are-relations point of view this can indeed be reasonable, as the fact that there exist relations to a certain entity should not automatically prevent that entity from being deleted; nevertheless, this means that the effect of a rule cannot be predicted precisely without knowing the host graph as well. Again, this may be avoided by including a test for the absence of such edges into the rule.

The closest alternative to simple graphs, which does not share the above characteristics, is the following.

Definition 2 (multigraph). *A multigraph is a tuple* $\langle V, E, src, tgt, lab \rangle$*, where* V *is an arbitrary set of nodes,* E *an arbitrary set of edges,* $src, tgt \colon E \to V$ *are source and target function and* $lab \colon E \to \mathsf{Lab}$ *an edge labelling function.*

Thus, in comparison to simple graphs, the source, target and labelling functions are now explicitly given rather than derived from E, and E itself is no longer a Lab-indexed binary relation but rather an arbitrary set. Thus, edges have their own identity, and parallel edges are automatically supported.

In the algebraic approach, multigraphs (in contrast to simple graphs) fall smoothly into the rich framework of adhesive categories and double-pushout rewriting. Among other things, this ensures that rule application is always reversible, which implies that the phenomena listed under the last two bullets above do not occur: if a rule adds an edge then the graph always changes under rule application (an edge is always added, possibly in parallel to an existing edge); and a node cannot be deleted unless all incident edges are explicitly deleted as well.

2.2 Nodification

Ordinary graphs, be they simple or multi, have very little structure in themselves. If they are used to encode models with rich structure, such as (for instance) design diagrams,

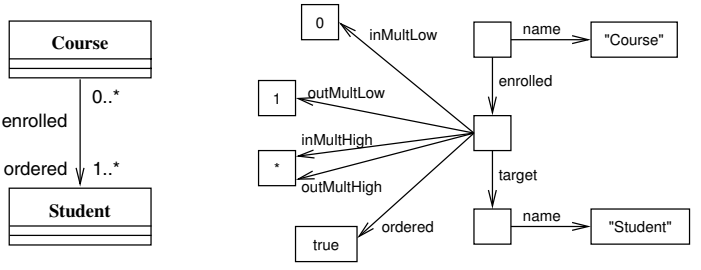


Fig. 1. A UML association type and its graph representation

then it may occur that the entity/relation intuition discussed above cannot be rigorously upheld. In particular, complex relationships, which should conceptually be represented by edges, may very well contain more information than what can be expressed by a single (binary) edge.

Example 1. For example, an association in a class diagram is conceptually a relation between two classes, but in addition to a name it typically has multiplicities and other modifiers (such as the fact that it is indexed or ordered). This is illustrated in Figure 1.

In such a case, the only solution is to introduce an additional node to capture the relation, with edges to the original source and target node, as well as additional edges to capture the remaining structure. We call this process *nodification*. Usually, one would prefer to avoid nodification, as it blows up the graphs, which has an adverse effect on visual appeal, understandability and complexity. This may be partially relieved by introducing a concrete syntax layer (syntactic graph sugar, such as edges with multiple labels) on top of the "real" graph structure, but that in itself also adds complexity to the framework.

Another context in which nodification occurs is in modelling relations among edges; for instance, if typing or traceability information is to be added to the graph. Rather than allowing "edges over edges", i.e., edges whose source or target is not a node but another edge, into the formalism itself, the typical solution is to nodify the edges between which a relation is to be defined.

Clearly, to minimise nodification it is necessary to move to a graph formalism with more inherent structure. A good candidate is *hypergraphs*.

2.3 Edges for Structure, Nodes for Glue

A hypergraph is a graph in which edges may have a different number of end nodes than two. Hypergraphs come in the same two flavours as binary graphs: simple and multi. However, especially multi-hypergraphs open up an intriguing alternative to the entity/relation intuition explored in Section 2.1, which has been exploited for instance in [2,3].[2]

[2] To complicate matters further, one can also extend simple graphs to n-ary rather than just binary relations. Though this is not a very popular model in graph transformation research, it is a faithful representation of, for instance, higher-degree relations in UML [55].

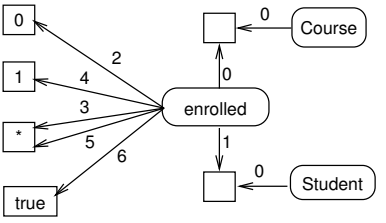


Fig. 2. Hyperedge representation of the right-hand graph of Figure 1

Definition 3 (hypergraph). *A hypergraph is a tuple* $\langle V, E, con, lab\rangle$*, where* $con\colon E \to V^*$ *is a connection function mapping each edge to its sequence of* tentacles*, and* $lab\colon E \to \mathsf{Lab}$ *is an edge labelling function. A node in a hypergraph is called* isolated *if it does not occur in the images of* con*.*

Usually, labels are used to determine the *arity*, i.e., the number of tentacles, of edges; that is, there is assumed to be an arity function $\alpha\colon \mathsf{Lab} \to \mathsf{Nat}$ such that $|con(e)| = \alpha(lab(e))$ for all $e \in E$. Though there is not an explicit notion of edge source, we can designate the first tentacle to be the source (which due to the possibility of nullary edges implies that src is a partial function).

Obviously, a binary (multi)graph is just a special case of a hypergraph in which all arities are 2. Edges with arity 0 and 1 are also potentially useful, to encode global state attributes and node labels, respectively. For instance, a 0-ary edge labelled Error (with no attached node) could be used to signify that the state is erroneous in some way; 1-ary edges Student (with source nodes only) can be used to label nodes that represent students. It is in the edges with higher arity, however, that extra structure can be encoded. For instance, the association edge of Figure 1 can be represented by a single, 7-ary edge, as shown in Figure 2. (In this figure we have followed the convention to depict hyperedges also as rounded rectangles, with numbered arrows to the tentacles. Thus, hyperedges look quite a bit like nodes.)

The shift in interpretation mentioned above lies in the fact that we may now regard edges to be the primary carrier of information, rather than nodes as in the entity/relation view. In particular, the entities can be thought of as represented by singular edges, like the Course- and Student-edge in Figure 2. The nodes then only serve to glue together the edges; in fact, they may be formally equated to the set of edge tentacles pointing to them. In particular, in this interpretation, an isolated node is meaningless and may be discarded ("garbage collected") automatically.

One of the attractions in this new interpretation is that it is no longer necessary to delete nodes: instead, it suffices to delete all incident edges, after which the node becomes isolated and is garbage collected. Thus, a major issue in the algebraic graph transformation approach becomes moot. (Formally speaking, for a rule that does not delete nodes, pushout complements are ensured to exist always.)

Example 2. One of the main sources of programming errors in C that is difficult to capture formally is the manual deallocation of memory. It is easy to erroneously deallocate memory cells that are still being pointed to from other parts of a program. However,

a graph transformation rule that attempts to delete such a cell will either remove all incident edges or be inapplicable altogether, depending on the way node deletion is handled. Neither captures the desired effect.

Using the interpretation discussed above (the glue/structure view, as opposed to the entity/relation view), however, this becomes straightforward to model: deallocation removes the singular edge encapsulating the value of a memory cell, but this does not give rise to node deletion as long as there is another ("stale") edge pointing to the node. However, any rule attempting to match such a stale edge and use its target node will fail (because the target node no longer carries the expected value edge) and hence give rise to a detectable error. The reallocation of improperly freed memory can be captured in this way as well.

Summarising: the special features of hypergraphs (coming at the cost of a more complex relation between edges and nodes) are:

- *Nodification can be avoided in many cases,* as hyperedges are rich enough to encode fairly complex structures.
- *Node deletion can be avoided altogether* if one follows the idea of garbage collecting isolated nodes, called the glue/structure interpretation above.

2.4 The Awkwardness of Attributes

Graphs are great to model referential structures, but do not model associated data of simple types (such as numbers, booleans and strings), usually called *attributes*, as conveniently. At first sight there is actually no problem at all: data values can easily be incorporated into graphs by treating them as nodes, and an attribute corresponds to an edge to such a data node. The problem is, however, that data values do not behave in all respects like nodes: they are not created or deleted, instead every data value always "exists" uniquely; moreover, the standard algebraic operations on these types can yield values from an infinite domain, which therefore formally must be considered part of every graph, even if it is not visualised.

Note that we have already implicitly used the representation of data values as nodes in Figures 1 and 2, which contain instances of natural numbers, booleans and strings.

In the last few years, the algebraic approach has converged on a formalisation (see [21]) in which data values are indeed special nodes, which are manipulated not through ordinary graph rules but through an extension relying on the standard algebras of the data types involved. Though satisfactory from a formal point of view, we still feel that the result is hybrid and conceptually somewhat awkward. This is to be regretted especially since competing formal verification techniques all concentrate on the manipulation of primitive data with very few higher-order structures (typically just arrays and records), which are then used to encode referential structures. This makes the comparison of strengths and weaknesses very skewed.

Edge attributes and nodification. Another issue is whether attributes should be restricted to nodes, or if edges should also be allowed to carry attributes. This is, in fact, connected to the choice of simple graphs versus multigraphs: only in multigraphs does the concept of edges with attributes make sense. Indeed, this capability in some cases

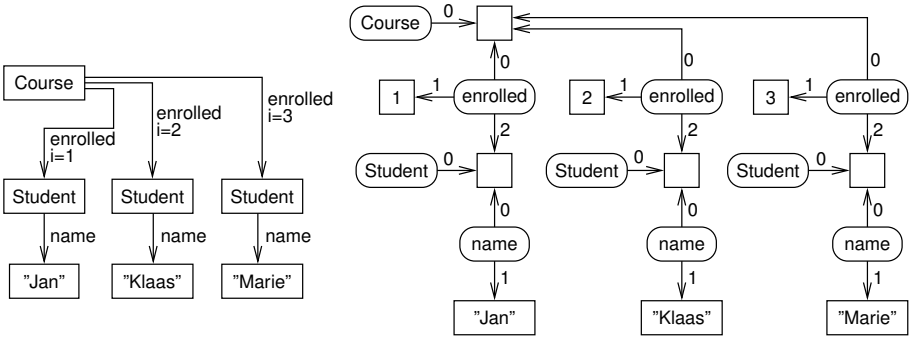


Fig. 3. Edge attributes or hyperedges for ordered edges

helps to avoid nodification, as we will show on an example below. However, the only way to capture edge attributes formally is to allow (attribute) edges to start at (non-attribute) edges. This makes the graph model more complex and less uniform, which feels like a heavy price to pay.

Example 3. A concept that comes up quite often in graph modelling is that of a list or sequence. For instance, the association shown in Figure 1 specifies that every Course has an associated *ordered* set of enrolled students. This means that an instance model may feature one Course with multiple associated Students, which are, moreover, ordered in the context of that Course.

One way to capture the ordering is to use indexing of the elements; and for this, edge attributes can be used. Apart from the label, every edge would receive a unique natural number-valued attribute, say i, as depicted by the left hand graph in Figure 3. The right hand side shows that the same effect can be achieved in hypergraphs.

The algorithmic alternative. The entire discussion above is driven by the intuition that attributes are just edges, albeit pointing to nodes with some special characteristics. Another point of view entirely is that attributes are distinct enough from edges to merit a different treatment altogether. For instance, attributes are not deleted and created, but are read and assigned; in the same vein, the concept of multiplicity typically does not apply to attributes.

In the algorithmic approach, there is in fact little objection to a separate treatment of edges and attributes: the latter are typically treated as local variables of a node. It is only in the algebraic approach that this meets objections. Depending on one's perspective, this can obviously be interpreted as a weakness of the algebraic approach, rather than the awkwardness of attributes.

Summarising: regarding attributes, we find that

– *The state-of-the-art algebraic formalisation of attributes feels awkward* and cannot compete with the ease of data manipulation in other verification techniques. On the other hand, there is no alternative that shares the same theoretical embedding, so if one values the algebraic approach then this is the only game in town.

– *Edge attributes come at a high conceptual cost,* which may be justified because they can help to avoid nodification. However, hypergraphs provide a more uniform way to achieve the same thing.
– *The algorithmic approach has no such difficulties* in incorporating attributes: they can be regarded as local variables of the nodes. This is a very pragmatic viewpoint, justification for which may be found in the fact that it reflects the practice in other modelling formalisms.

2.5 To Type or Not to Type

The graphs presented up to this point are not restricted in the way edges or edge labels can be used. In practice, however, the modelling domain for which graphs are used always constrains the meaningful combinations of edges. For instance, in the example of Figure 3 it would not make sense to include enrolled-edges between Student-nodes.

Such constraints are usually formulated and imposed through so-called *type graphs*. A type graph — corresponding to a *metamodel* in OMG terminology, see [54] — is itself a graph, typically in the same formalism as the object graphs, but often with additional elements in the form of inheritance and multiplicities (a formal interpretation of which can be found in [5, 80]). In the presence of a type graph, a given object graph G is only considered to be correct if there exists a structure-preserving mapping τ, a *typing*, to the type graph T. Structure preservation means (among other things) that τ maps G-nodes to T-nodes and G-edges to T-edges.

In fact, we can include the typing itself into the definition of a graph. If, in addition, we select the nodes and edges of the type graph T from the set of labels (i.e., $V_T \cup E_T \subseteq \mathsf{Lab}$), then the typing subsumes the role of the edge labelling function: τ assigns a label not only to every edge of G but to every node as well.

Inheritance. Normally, a typing must map every edge consistently with its source and target node: that is, $\tau(src_G(e)) = src_T(\tau(e))$ and $\tau(src_G(e)) = src_T(\tau(e))$ for all $e \in E_G$. However, this can be made more flexible by including node inheritance. Node inheritance is typically encoded as a partial order over the nodes of the type graph. That is, there is a transitive and antisymmetric relation $\sqsubseteq \subseteq V_T \times V_T$; if $v \sqsubseteq w$ then we call v a subtype of w. Now it is sufficient if $\tau(src_G(e)) \sqsubseteq src_T(\tau(e))$ and $\tau(src_G(e)) \sqsubseteq src_T(\tau(e))$.

Since the purpose is to ensure that we are only dealing with correctly typed graphs, the matching as well as the application of transformation rules has to preserve types. Without going into formal detail (for which we refer to [5]), the intuition is that the type of a node in the rule's left hand side must be a supertype of the type of the matching host graph node. Thus, the rule can be defined abstractly (using general types) and applied concretely (on concretely typed graphs).

Example 4. Figure 4 shows a type graph (on the left), where the open triangular arrow visualises the subtyping relation $\mathsf{B} \sqsubseteq \mathsf{A}$. Of the four graphs on the right, both (i) and (ii) are correctly typed, and if (i) is the left hand side of a rule then it has two valid matches into (ii). (iii) is incorrect because the source type of the b-edge (A) is not a subtype of b's source (B).

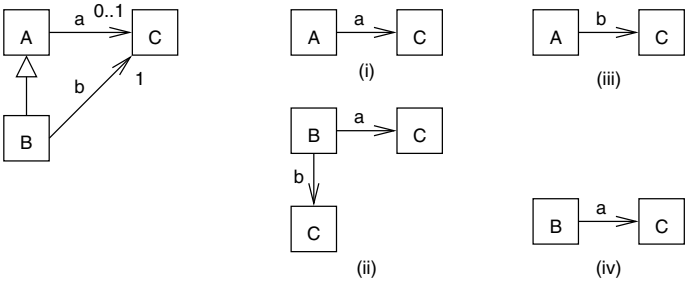


Fig. 4. A type graph (on the left) with correctly typed graphs (i) and (ii), and incorrectly typed graphs (iii) and (iv)

The above deals with node type inheritance only. In addition one may also consider a subtype or inheritance relation over edges. There is far less consensus on the usefulness and, indeed, meaning of edge type inheritance; see, for instance, [80] for one possible formalisation. In [42] we have shown that UML has three different notions of edge subtyping (subsets, redefinitions and unions). We therefore think of edge subtyping as a kind of special constraint, some more examples of which are briefly reviewed below.

Multiplicities. Also very common in type graphs are edge multiplicities. For instance, Figure 4 shows outgoing edge multiplicities, expressing that every A-node has at most one outgoing a-edge, and every B-node has *exactly* one outgoing b-edge. Graph (iv) is incorrect because it does not satisfy the multiplicity constraint on the b-edge.

Multiplicities can be attached both to outgoing and to incoming edges, and they specify "legal" ranges for the number of such edges in an object graph. In contrast to node inheritance, however, in general it is not decidable whether all graphs that can be constructed by a given transformation system are correct with respect to the edge identities. In such cases, the multiplicity constraint must be regarded as a property of which the correctness must be established through further analysis.

Special constraints. In general, as we have stated at the beginning of this subsection, typing is a way to impose constraints on graphs. Inheritance allows to refine these constraints, and multiplicities provide a way to strengthen them. However, the story does not end here: in the course of the years many special types of constraints have been proposed that can be seen as type enrichments. One has only to look at the UML standard [55] to find a large collection of such special constraints: abstractness, opposition, composition, uniqueness, ordering, as well as the edge subtype-related notions already mentioned above. An attempt to categorise these and explain them in terms of the constraints they impose can be found in [42]. Ultimately one can resort to a logic such as OCL to formulate application-specific constraints. It is our conviction that, where node inheritance and multiplicities have long proved their universal applicability and added value, one should be very reticent in adopting further, special type constraints.

Summarising: Regarding type graphs, we find that

– *Type graphs impose overhead on the formalism* as they have to be created, checked and maintained while creating an actual transformation system. Small prototype

graphs and rules can be developed faster if it is not a priori necessary to define a type graph and maintain consistency with it.
– *Type graphs strengthen the formalism* as they allow to document and check representation choices and provide a straightforward way to include node labels. Moreover, inheritance is a convenient mechanism for the generalisation of rules.

2.6 Special Graphs for Special Purposes

The classes of graphs we have discussed above impose very few artificial restrictions on the combinations of nodes and edges that are allowed. In fact, the only restriction is that in simple graphs parallel edges are disallowed. This lack of restrictions is usually considered to be part of the appeal of graphs, since it imparts flexibility to the formalism. (Of course, type graphs also impose restrictions, but those are user-defined rather than imposed by the formalism.)

However, there is a price to pay for this flexibility, in terms of time and memory: the algorithms and data structures for general graphs are more expensive than those for more dedicated data structures such as trees and arrays. If, therefore, the modelling domain at hand actually does not need this full generality, it makes sense to restrict to special graphs. Here we take a look at *deterministic* graphs.

Definition 4 (determinism). *We call a graph G* deterministic *if there is a* Lab*-indexed family of partial functions $(f_a \colon N_G \rightharpoonup E_G)_{a \in \mathsf{Lab}}$, such that*

$$lab_G(e) = a \wedge src_G(e) = v \;\Leftrightarrow\; f_a(v) = e \tag{1}$$

In other words, a graph is deterministic if, for every label a, every node has at most one outgoing a-labelled edge. Note that this is meaningful for all types of graphs we have discussed here (even hypergraphs) as they all have edge sources and edge labels. In terms of type graphs (Section 2.5), determinism corresponds to the specification of outgoing edge multiplicity 1 for all edges.

Advantages of determinism. Deterministic graphs offer advantages in time and memory consumption:

– There are cheaper data structures available. For instance, every node we can keep a fixed-sized list of outgoing edges, with one slot per edge label, holding the target node identity. The slot defaults to a null value if there is no outgoing edge with that label. For general graphs, instead we have to maintain a list of outgoing edges of indeterminate length.
– There are cheaper algorithms available. For instance, matching an outgoing edge will not cause a branch in the search space, as has been investigated in [16]. The same point has been made in [17], where so-called V-structures are introduced which share some characteristics of deterministic graphs.

Deterministic graphs in software modelling. In the domain of software modelling, edges typically stand for *fields* of structures or objects. Every field of a given object holds a value (which can be a primitive data value or a reference to another object).

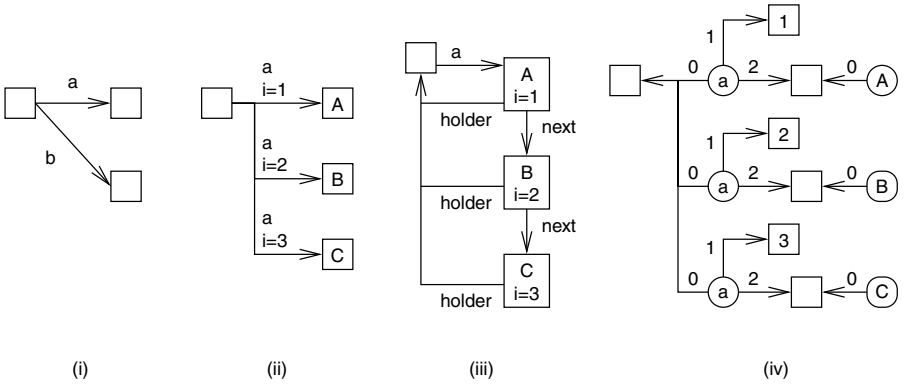


Fig. 5. (i) is a deterministic graph representing an object with fields a and b; (ii) is a non-deterministic edge-attributed graph modelling an array a; (iii) is a deterministic node-attributed graph, and (iv) a δ-deterministic hypergraph modelling the same array.

Such fields have names that are unique in the context of their holder (the structure or object); the field names are used as edge labels. Thus, the resulting graphs are naturally deterministic.

On the other hand, another common data structure is the *array*, which has indexed slots also holding values. Deterministic graphs are not very suitable to model such structures: the natural model of an array with local name a is a node with multiple outgoing a-edges, one for each slot of the array. A deterministic graph representation can be obtained by using a linked-list principle, possibly with backpointers. Both solutions are shown in Figure 5.

Maybe the most satisfactory solution is to use hypergraphs (in which the index is one of the tentacles, as in Figure 3) and broaden the concept of determinism.

Definition 5 **(δ-determinism).** *Let* $\delta\colon \mathsf{Lab} \to \mathsf{Nat}$ *be a* degree *function such that* $\delta(a) \leq \alpha(a)$ *for all* $a \in \mathsf{Lab}$. *A hypergraph* G *is called* δ*-deterministic if there is a* Lab*-indexed family of functions* $(f_a\colon V_G^{\delta(a)} \to E_G)_{a\in\mathsf{Lab}}$ *such that for all* $e \in E_G$*:*

$$lab_G(e) = a \wedge con(e)\!\downarrow_{\delta(a)} = \boldsymbol{v} \;\Leftrightarrow\; f_a(\boldsymbol{v}) = e$$

Here $\boldsymbol{v}$ denotes a sequence of nodes, and $-\!\downarrow_i$ for $i \in \mathsf{Nat}$ projects sequences to their first i elements. Thus, edges are no longer completely determined by their label and source node, but by their label and an initial subsequence of their tentacles (of length given by δ). Though this needs further investigation, we believe that some of the advantages of (strictly) deterministic graphs can be salvaged under this more liberal notion.

The standard notion of determinism (Definition 4) corresponds to degree 1 ($\delta(a) = 1$ for all $a \in \mathsf{Lab}$); graph (iv) in Figure 5 is δ-deterministic for $\delta(\mathsf{a}) = 2$ (and all other δ-values equal to 1).

2.7 The Pushout Scare

We feel it to be necessary to devote some words to an issue that, in our opinion, stands in the way of a broader acceptance of graph transformation, namely the intimate and inescapable connection of the algebraic approach to category theory. The only generally understood and accepted way to refer to the mainstream algebraic approach is the *double pushout* approach; the main alternative is called the *single pushout* approach. This automatically pulls the attention to the non-trivial pushout construction, a proper understanding of which requires at least a week of study to grasp only the bare essential concepts from category theory.

For those who stand above this theory, having mastered it long ago, it may be difficult to appreciate that this terminology repels potential users. In order to give more of an incentive to researchers from other areas to investigate graph transformation as a potentially useful technique, an alternative vocabulary must be developed, which does not suggest that knowing category theory is a prerequisite for using graph transformation. We propose the following:

Conservative graph transformation as an alternative term for the double pushout-approach. The term "conservative" refers to the fact that the deletion of graph elements carries some implicit application conditions, namely that all incident edges of deleted nodes are also explicitly deleted at the same time, and that the nodes and edges a rule attempts to delete are disjoint from those it intends to preserve. Thus, a rule may fail to be applicable even though a match exists.

Radical graph transformation as an alternative term for the single pushout-approach. The term "radical" refers to the fact that rules are always applicable (in a category where all pushouts exist, as they do for simple graphs) but the effect may go beyond what the rule explicitly specifies: dangling edges are deleted, and case of non-injective matches, nodes and edges may deleted that the rule appears to preserve.

2.8 Summary

In Table 1 we have summarised the five dimensions of the graph zoo discussed above. The dimensions are orthogonal: all combinations can be defined — however, some combinations make more sense than others. For instance, edge attribution in hypergraphs does not seem very useful, as additional tentacles provide a more uniform way to achieve the same modelling power. Other unlikely combinations are node attribution for hypergraphs, and edge attribution for simple graphs. Finally, note that the distinction between simple and multigraphs disappears if we restrict to deterministic graphs.

It should be noted that, again, we do not claim comprehensiveness of this overview. One dimension that we have omitted altogether is the notion of hierarchy, which by many researchers is considered to be an important enough notion to deserve a direct encoding into graphs; see, for instance [19,56], but also the distributed graphs of Taentzer [79,76] and Milner's bigraphs [51,52]. No doubt there are other dimensions we have missed.

Table 1. Dimensions of the graph zoo

Dimension	Value	Meaning	Advantages
Edge encoding	*simple*	no edge identity	edges are relations; simple
	multi	edges have own identity	better algebraic properties
Arity	*standard*	binary edges only	uniform, simple
	hyper	arbitrary tentacle count	avoids nodification
Attribution	*none*	no data attributes	simple
	nodes	only node attributes	necessary in practice
	all	node and edge attributes	avoids nodification
Typing	*no*	no explicit typing	simple, no overhead
	yes	type graph	documentation, node labels, inheritance
Determinism	*no*	general graphs	flexible
	yes	deterministic graphs	more efficient

3 Criteria for Behavioural Specification and Analysis

We will now step through a number of criteria for any formalism that is intended for specifying and analysing system behaviour. Behavioural analysis, in the sense used in this paper, at least includes some notion of state space exploration, be it partial, complete, abstract, concrete, guided, model checked, explicit, symbolic or a combination of those. Graph transformation tools that offer some functionality of this kind are, for instance, AUGUR2 [44], FUJABA [30] and GROOVE [58]; we want to stress, however, that we are more interested in the potential of the formalism than in actual tools.

As announced in the introduction, we will make a distinction based on whether the formalism is to be directly used as a modelling language, or as an underlying semantics for another language. For each of the criteria, we judge how well graph transformation does in comparison to other techniques — where we are thinking especially of SPIN [38], a state-of-the-art explicit-state model checker that uses a special-purpose language Promela for modelling.

3.1 Learning Curve

This term refers to the expected difficulties in learning to read and write specifications in a formalism or calculus. Another term might be "conceptual complexity." This in itself depends on a number of factors, not the least of which is editor tool support; but also general familiarity with the concepts of the formalism is important. Thus, a language like Promela, with concepts that are at least partly familiar from everyday programming languages, has a shallower learning curve than languages that heavily rely on more esoteric mathematical concepts.

The learning curve of graph transformation. Given adequate tool support for graph editing (which most graph transformation tools offer), the visual nature of specifications can appeal to intuition and make the first acquaintance with the formalism easy. However, there are several complications that require a more thorough understanding from the user, prime among which is the treatment of data attributes — see Section 2.4 for a more extensive discussion about this.

In general, we believe that the learning curve is directly related to the simplicity of the graph formalism: see Table 1 for our evaluation on this point. In particular, we have observed many times that novice users are *very fast* in creating prototype specifications (in a wide variety of domains) using GROOVE [58], which has about the simplest graph formalism imaginable. Hypergraphs, on the other hand, although offering several benefits (as argued in the previous section) definitely have a steeper learning curve.

Though we have complained above (Section 2.7) about the intimate connection with category theory, actually for the use of graph transformation in specifications an understanding of the underlying theory is not necessary; so we find that this does not put a burden on the learning curve.

Relevance. The learning curve of a formalism is, of course, very relevant if it is to be used directly as a modelling language: all users then have to be versed in reading and writing specifications in that language. However, the learning curve is less relevant if the formalism is to be used to define operational semantics. In the latter case, only a limited number of users will have to understand the formalism, namely those who have to understand or implement the semantics; and even fewer will have to write in it, namely those who define the semantics.

3.2 Suitability

It seems very obvious that a formalism to be used for specifying the behaviour of software systems should be especially suitable for capturing the specific features of software. For instance, the dynamic nature of memory usage (both heap and stack) and the referential (object) structures that are ubiquitous in most modern software should be natively supported by the formalism.

Yet it turns out that even formalisms that are not at all suitable in this sense, for instance because they only support fixed-size or very primitive data structures (like Promela), nevertheless do quite well in software verification. A prime example is the use of SAT-solvers for (bounded) software model checking (see [49,40]): the formulae being checked are extremely primitive and only finite behaviour can be analysed, but because the tools can deal with truly enormous datasets (they are very scalable, see Section 3.4 below), the results are quite good.

Suitability of graph transformation. Precisely the dynamic nature of software and its stress on referential structures make graph transformation an excellently suitable formalism. Indeed, this is one of it strongest points.

Relevance. Though, as pointed out above, there is a balance between suitability and scalability, there is no question that suitability (in the sense used here) is a very relevant criterion in judging a formalism.

3.3 Flexibility

Flexibility refers to the ease of modelling an arbitrary problem using a given formalism, without having to resort to "coding tricks" to make it fit. That is, if the model captures the problem without introducing extraneous elements, we call it suitable for

the modelling domain (as in Section 3.2 above), whereas if it does so for a broad range of domains, we call it flexible.

Flexibility of graph transformation. The flexibility of graph transformation is another of its strong points. Indeed, besides software analysis we have seen uses for GROOVE in the context of network protocol analysis, security scenarios, aspect-oriented software development, dynamic reconfiguration and model transformation.[3]

Relevance. In the dedicated domain of behavioural specification and analysis, the (general) flexibility of a formalism is less important than its (specific) suitability. Still, if the formalism is directly to be used as a modelling language, the ability to model concepts natively is very attractive. If, on the other hand, it is to be used for operational semantics, the importance of being flexible goes down, as the domain of application is then quite specialised.

3.4 Scalability

Scalability refers to the ability of a formalism to deal with large models. We typically call a formalism highly scalable if its computational complexity is a small polynomial in the model size. However, another notion of scalability refers to the *understandability* of larger models, i.e., their complexity to the human mind. We call this "visual scalability."

Visual scalability. Though small graphs (in the order of ten to twenty nodes) are typically easy to grasp and may provide a much more concise and understandable model than an equivalent textual description, this advantage is lost for larger graphs. Indeed, as soon as graphs start to be non-planar, even the best layouting algorithms cannot keep them from becoming very hard to understand. The typical solution in such a case is to decompose the graph into smaller ones, and/or to resort to a more text-based solution by introducing identifiers instead of edges.

Computational scalability. This is another term for computational or algorithmic complexity, which is a huge field of study on its own. In the context of behavioural analysis, algorithmic complexity is of supreme importance, as the models tend to become very large — this is the famous "state space explosion problem," which refers to the fact that state space tends to grow exponentially in the number of independent variables and in the number of independent components. Virtually all of the work in model checking is devoted to bringing down or cleverly representing the state space (through abstraction, compositionality or symbolic representation) or improving performance (through improved, concurrent algorithms).

Scalability of graph transformation. In general, graph transformation does not scale well, either visually or computationally. The former we have already argued above; for the latter, observe that the matching of left hand sides into host graphs, which is a

[3] As an aside, it should be mentioned that the flexibility of the graph transformation formalism does not only lie in the graphs themselves but also in the transformation rules and their composition into complete system specifications. In the current paper we have decided to omit this aspect altogether.

necessary step in rule application, is an NP-complete problem in the size of the left hand side, and polynomial in the size of the host graph where the size of the LHS (regarded as a constant) is the exponent. However, there are two alleviating factors:

- By using incremental algorithms, the complexity of matching becomes constant, at the price of more complex updating (i.e., part of the problem is shifted to the actual transformation). In [9] it is shown that this improves performance dramatically. Investigations in the context of verification are underway.
- In modelling software behaviour, typically matches do not have to be *found*: they are already determined by the context — as should be expected, given that programming languages are mostly deterministic (except for the effects of concurrency). In this setting, the complexity of matching may also be constant time: see [16].

In the context of state space exploration, another issue is the identification of previously encountered states. In general, this involves isomorphism checking of graphs, which is another (probably) non-polynomial problem. However, in [59,15] it has been shown that in practice isomorphism checking is feasible, and may give rise to sizable state space reduction in case the model has a lot of symmetry.

Relevance. Visual scalability is very relevant if the users need to visualise large models. Unless good decomposition mechanisms are available, this is indeed the case when the formalism is used as a modelling language, but less so if it is used for operational semantics.

As indicated above, computational scalability is supremely important, regardless of the embedding of the formalism.

3.5 Maturity

With maturity we refer to the degree in which all questions pertaining to the use of a formalism in the context of behavioural specification and analysis have been addressed and answered. We distinguish theoretical and practical maturity.

Theoretical maturity. We have already referred above (see Section 3.4) to the many existing techniques to combat the state space explosion problem: compositionality, abstraction and symbolic representations. We call a formalism theoretically mature if it is known if and how these techniques are applicable in the context of the formalism.

Practical maturity. Practical maturity, in this context, refers to the degree to which algorithms and data structures enabling the actual, efficient implementation of the formalism and corresponding analysis techniques have been investigated. Thus, we call a formalism practically mature if there is literature describing and evaluating such algorithms and data structures.

Maturity of graph transformation. Despite the impressive body of theoretical results in algebraic graph rewriting (recently collected in [22]), we think the formalism is not yet theoretically mature for the purpose of behavioural specification and analysis. Many of the concepts mentioned under Section 3.4 have hardly been addressed so far.

– *Abstraction.* Without some form of abstraction, it will never be possible to analyse arbitrary graph transformation systems, as these are typically infinite-state. Since it is not even decidable whether a given graph transformation system is finite-state, this is a severe restriction. Thus, we regard the existence of viable abstraction techniques as an absolute necessity for graph transformation to realise its full potential as a technique for behavioural specification and analysis. Consider: if all we can analyse is a finite, bounded fragment of the entire state space, then there is no advantage over existing model checking techniques in which the infinite-state behaviour cannot even be specified.

  The only fully worked out approach for abstraction, apart from several more isolated attempts described in [61,7], is the abstraction refinement work of [43], which is part of the Petri graph approach briefly mentioned in the discussion of hypergraphs (Section 2.3). Although the results are impressive, the rule systems that can be analysed are rather specialised (for instance, only rules that delete all edges of the left hand side are allowed). There is much more potential for abstraction techniques, as for instance the work on shape analysis (e.g., [66]) shows.
– *Compositionality.* This is another effective technique to combat state space explosion, which finds its origin in process algebra (see, e.g., [50]). A formalism is compositional if specifications of large, composed systems can be themselves obtained by composing models of subsystems (for instance, parallel components or individual objects). Those smaller models can then be analysed first; the results can be used to draw conclusions about the complete system without having to construct the entire state space explicitly.

  Graph transformation is, by nature, non-compositional. Instead it typically takes a whole-world view: the host graph describes the entire system. Though several approaches have been studied to glue together rules, most notably the work on rule amalgamation [75] (with the offshoot of distributed graph transformation in [79]), the best studied approach we know in which the host graph may be only partially known is the *borrowed context* work of [4]. This, however, falls short of enabling compositionality, since the host graph always grows, and never shrinks, by including a borrowed context; thus this does not truly reflect the behaviour of a single component. A very recent development on compositional graph transformation is [60].
– *Symbolic representation.* This refers to the implicit representation of a state space through a formula that holds on all reachable states and fails to hold on all unreachable ones. It has been shown that, by representing such formulae as Binary Decision Diagrams (BDDs), an enormous reduction with respect to an explicit state space representation can be achieved; see, e.g., [10].

  To the best of our knowledge, there have been no attempts whatsoever to develop similar symbolic representations for graphs. Clearly, the difficulties are great, as one of the main heuristics in the success of BDDs is finding a suitable ordering of the state vector; it is not at all easy to see how this translates to graph transformation, where the states are graphs which do not give rise to an unambiguous linearisation as vectors. For another thing, the graphs are not a priori bounded, which is a requirement for standard BDDs. Nevertheless, observing the success of BDDs in state-of-the-art model checking, they are certainly worth a deeper investigation.

As for practical maturity, fairly recently there has been increased attention for fast algorithms and data structures, especially for graph matching; for instance, see [32,39,9]. Another issue is isomorphism checking, which was studied in [59]. Furthermore, below we report on tool support for graph transformation, in the context of which there are continuous efforts to improve performance, stimulated by a series of tool contests.

Relevance. As we have indicated above, we believe that the theoretical maturity of the field is one of the prime criteria for the success of graph transformation for behavioural specification and analysis. It does not make a difference whether the formalism is to be used directly as a specification language or as an underlying operational semantics. As for practical maturity, this is also important, but it partially follows the theoretical insights.

3.6 Tooling

The final criterion we want to scrutinise is tooling. We discuss two aspects.

Tool support. In order to do anything at all with graph transformation in practice, as with any other formalism it is absolutely necessary to have tool support, in the sense of individual tools to create graphs and rules and perform transformations. Such tools should satisfy the normal usability requirements; in particular, they should be well enough supported for "external users" to work with them, and there should be some commitment to their stability and maintenance.

Standardisation. If we want to employ tools for graph transformation in a larger context, where the results are also used for other purposes, it is vital to have a means to communicate between tools. This requires a measure of interoperability, which in turn requires standardisation.

In fact one may distinguish two types of interoperability: one involves the interchange of graphs and rules, and the other involves communicating with other tools that are not graph-based. For the second type, the main issue is to adhere to externally defined formats; this can be solved on an individual bases by each tool provider. We will concentrate on the first type, which is a matter for the graph transformation community as a whole.

Tooling for graph transformation. Starting with PROGRES [69] and AGG [29] in the mid-80s, tool support for graph transformation has become increasingly available. We know of at least ten groups involved in a long-lasting, concerted effort to produce and maintain generally usable tools for graph transformation. Moreover, recently we have started a series of tool contest with the primary aim of comparing existing tools, and the secondary aim of creating a set of available case studies which can be used to try out and compare new tools or new functionality; see [62,63,48]. All this points to an extensive investment in tool support for graph transformation in general.

On the other hand, for the more specific purpose of behavioural specification and (especially) analysis, we find that the situation is less encouraging. The main special feature that is needed in this context is the ability to reason about all reachable graphs,

usually by exploring and analysing the state space systematically, rather than to efficiently apply a single sequence of rules (possibly with backtracking). Some (relatively) early work exists in the form of CHECKVML [68] and Object-Based Graph Grammars [18], both of which rely on a translation to the standard model checking tool SPIN [38] for the state space exploration. Our own tool GROOVE [58] has a native implementation of this functionality. A more recent development is AUGUR2 [44], which uses advanced results from Petri net theory to create a finite over-approximation (called an unfolding), already mentioned in the discussion on abstraction in Section 3.5, which can also cope with infinite state spaces. A more small-scale effort, also based on over-approximation, is reported in [8]. Yet another approach, based on backwards exploration and analysis, is implemented in the tool GBT [31]; see [67] for some results. Finally, [57] uses assertional (weakest-precondition) reasoning rather than state space exploration, which is also capable of dealing with infinite-state systems.

The (relative) scarcity of graph transformation tools for analysis is borne out by the fact that, in the last transformation tool contest [48], the "verification case" (concerning the correctness of a leader election protocol) received only 4 submissions.

The situation with respect to standardisation is not so good. In the past there have been serious attempts to define standards for the interchange of graphs (GXL , see [37]) and of graph transformation systems (GTXL , see [77,46]), both of which are XML-based standards; however, an honest assessment shows that there has not been much much activity of late, and the standards are not supported by many tools. We do not know of a single instance of such a standard being used to communicate between different tools.

A similar point is made in [34], where it is argued that the level on which GTXL has attempted to standardise is too low-level.

Relevance. It goes without saying that tool support is absolutely necessary for the success of graph transformation, if it is to be a viable formalism for behavioural specification and analysis. It may be less immediately obvious that the same holds for standardisation, in the sense used here (restricted to graph and rule standards). Yet it should be realised that there is a multitude of mature tools around, most of which are specialised towards one particular goal. A common, agreed-upon standard would enable tool chaining in which the strengths of each individual tool can be exploited to the maximum.

3.7 Summary

In Table 2 we summarise the results of this section, by scoring graph transformation on the criteria discussed above, and also rating the relevance of the criteria. We have used a scale consisting of the following five values, with a slightly different interpretation in the first two and the last two columns:

−− "Very weak", respectively "irrelevant."
− "Meagre", respectively "less relevant."
0 "Reasonable", respectively "partially relevant."
+ "Good", respectively "relevant."
++ "Excellent", respectively "very relevant."

Table 2. Summary of criteria

Criterion		General graphs	Special graphs	Other techniques (SPIN)	Relevance for use as modelling language	Relevance for use as operational semantics
Learning curve		0	0	+	+	−
Suitability		++	+	−	+	0
Flexibility		++	0	−	+	−
Scalability	Visual	−−	−		+	−−
	Computational	−	+	++	++	++
Maturity	Theoretical	0	0	++	++	++
	Practical	+	+	++	+	+
Tooling	Support	+	+	++	++	+
	Standardisation	−	−	−	+	++

We have divided the criteria into those that are intrinsic to the formalism of graph transformation (the top five) and those that can be improved by further effort (the bottom four). We stress again that the scores are subjective.

Special graphs. The column "special graphs" in the table refers to the possibility of limiting graphs to a subclass with better computational properties. Here we are thinking especially of deterministic graphs (see Section 2.6). In Section 3.4 we have already cited [16] who point out that matching can be much more efficient for such graphs, in particular if it is already known where in the graph the next transformation is to take place, as is often the case when modelling software. Further evidence for the increased performance for special graphs is provided by the tool FUJABA [30], where the graph formalism is tuned for compilation to Java. For instance, the Sierpinski case study described in [78] shows that this can give rise to extremely high-performance code.

Other techniques. As stated at the start of this section, we have taken SPIN (see [38]) as a prototypical example of an "other technique" for the specification and analysis of system behaviour, because it is a well-known, mature tool that is actually being used successfully in practice. To our opinion, the qualities of SPIN are rather complimentary to those of graph transformation: it is not at all trivial to encode the desired behavioural aspects of a system into Promela, which is built upon a fixed set of primitives such as threads, channels and primitive data domains; however, once this task is done, the tool performs amazingly well. Surprisingly, SPIN , too, suffers from a lack of standardisation; it gets by because Promela itself is, in effect, an "industrial" standard.

Evaluation. There is no single strong conclusion to be drawn from this table. Here are some observations:

– Two of the criteria on which graph transformation does especially well, namely suitability and flexibility, are much less relevant when using graph transformation as an underlying semantics than when using it as a modelling language. The same holds for a criterion in which graph transformation does especially badly, namely visual scalability.

– Though theoretical maturity and standardisation are rated as average or worse, improving this is a matter of further research; the poor scalability, on the other hand, is intrinsic to graphs.
– The improved computational scalability of deterministic graphs may be enough to prefer those over arbitrary graphs, in the context of behavioural specification and analysis, also given that their suitability is hardly less than for general graphs. This is especially true if the formalism is to be used for operational semantics, since there the loss of flexibility is less relevant.

4 Recommendations

In this section we formulate a series of personal recommendations, based on the observations made in this paper.

The right graphs. Though the overview in Section 2 is primarily intended (as the section title says) to provide a roadmap through the possibly bewildering set of graph variations, one can also use it for another purpose, namely to select the "right" graph formalism, meaning the kind of graphs with the most advantages.

For us, the winner is hypergraphs (Section 2.3), based on the following observations:

+ Hypergraphs make nodification (see Section 2.2) unnecessary in many cases. Thus, graphs can remain simpler (in terms of node and edge count), which directly helps visual and computational scalability.
+ Using hypergraphs, node deletion can be avoided; instead, nodes can be "garbage collected" whenever they have become isolated. Thus, in terms of the algebraic approach, all pushout complements exist: in other words, the only condition for rule applicability is the existence of a matching.
+ Hyperedges can combine data-valued and node-values tentacles, and thus provide an elegant extension to attributed graphs.
− The variable arity of hyperedges is an added complication, both for the development of theory and for tool support. (We believe that this is outweighed by the fact that nodification can often be avoided.)
− The usual graphical convention for hypergraphs, where edges are depicted as node-like boxes and their tentacles as arrows from those to the "real" nodes, is cumbersome and unattractive. (We believe that alternative, more intuitive notations can be devised.)

Terminology. In Section 2.7, we have made some recommendations for alternatives to the scary category theoretical terminology of algebraic graph transformation: *conservative transformation* for the double pushout approach, and *radical transformation* for the single pushout approach. Even if these suggestions are not themselves taken up, we hope to inspire a discussion on this point.

Theoretical maturity. In Section 3.5 we have identified several open issues, the solution of which we believe to be of prime importance for the success of graph transformation in the area of behavioural specification and analysis. To reiterate, they are *abstraction*,

compositionality and *symbolic representation*. No doubt there are more such issues; however, we propose to address these in the coming years.

Standardisation. Standardisation efforts in the past have led to an XML-based standard for graph exchange, GXL (see [37]) and a related standard for the exchange of graph transformation systems, GTXL (see [77,46]). However, as noted in Section 3.6, these standards have not made as much impact as one would wish.

A good standard can bring many benefits:

- Each individual tool or application can use the standard for storing its own data structures; there is no need for repeating the difficult process of defining a storage format. (Note that this is only a benefit if the standard does not impose overhead that is unnecessary from the perspective of the tool or application at hand.)
- In the same vein, standards can be used as a basis for the exchange of data between different tools; in other words, for the interoperability of tools. This, in fact, is typically the main driving force behind the definition of standards. However, interoperability is not automatically achieved even between tools that use a given standard for storage, as they may fail to support the *entire* standard. Also, the task of making a tool compliant to a standard is one that will be undertaken only if the concrete benefits are clear; in practice it turns out that this is only rarely the case.
- If standards are human-readable, rather than just digestible to computers, then they may serve understandability. A person would need only to get versed in one notation, used across formalisms or tools, to easily read and interpret all models and specifications. Note, however, that the requirement of human-readability rules out XML-based formats.

We propose to revive the standardisation effort; if not on the grand scale of GXL and GTXL (which essentially have the ambition of encapsulating arbitrary graphs and graph transformation systems), then possibly among a small set of tools whose creators share the willingness to invest the necessary effort.

Cooperation. The last point we wish to make does not follow from any of the discussion in this paper, but is rather a call, or challenge. From approximately 1990 onwards, there has been almost continuous European level funding for graph transformation-based research, in the form of COMPUGRAPH [20], SEMAGRAPH [74], APPLIGRAPH [1], GETGRATS [33], SEGRAVIS [72] and SENSORIA [73]. Though some of these projects have a more theoretical focus and others a more practical, there seems little doubt that they have together fostered strong ties within the community and benefited both theory and practice of graph transformation.

However, the last of these projects (SENSORIA) is due to end in 2010. It is therefore high time to take a new initiative. As should be clear from this paper, we strongly feel that only by combining the best of theory and practice we can progress further. At least in the area of behavioural specification and analysis, graph transformation will only be able to prove its added value if we do not hesitate to tackle the open theoretical issues and at the same time continue working on tool support and standardisation.

Who will take up this gauntlet?

References

1. APPLIGRAPH: Applications of graph transformation (1994), http://www.informatik.uni-bremen.de/theorie/appligraph/
2. Baldan, P., Corradini, A., König, B.: A static analysis technique for graph transformation systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 381–395. Springer, Heidelberg (2001)
3. Baldan, P., Corradini, A., König, B.: A framework for the verification of infinite-state graph transformation systems. Inf. Comput. 206(7), 869–907 (2008)
4. Baldan, P., Ehrig, H., König, B.: Composition and decomposition of dpo transformations with borrowed context. In: [13], pp. 153–167
5. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 214–228. Springer, Heidelberg (2004)
6. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: [24], pp. 431–433
7. Bauer, J., Boneva, I., Kurbán, M.E., Rensink, A.: A modal-logic based graph abstraction. In: [25], pp. 321–335
8. Bauer, J., Wilhelm, R.: Static analysis of dynamic communication systems by partner abstraction. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 249–264. Springer, Heidelberg (2007)
9. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: [25], pp. 396–410
10. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Logic in Computer Science (LICS), pp. 428–439. IEEE Computer Society, Los Alamitos (1990)
11. Ciraci, S.: Graph Based Verification of Software Evolution Requirements. PhD thesis, University of Twente (2009)
12. Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: [24], pp. 383–398
13. Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.): ICGT 2006. LNCS, vol. 4178. Springer, Heidelberg (2006)
14. Corradini, A., Montanari, U. (eds.): Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA). Electronic Notes in Theoretical Computer Science, vol. 2 (1995)
15. Crouzen, P., van de Pol, J.C., Rensink, A.: Applying formal methods to gossiping networks with mCRL and GROOVE. ACM SIGMETRICS Performance Evaluation Review 36(3), 7–16 (2008)
16. Dodds, M., Plump, D.: Graph transformation in constant time. In: [13], pp. 367–382
17. Dörr, H.: Efficient Graph Rewriting and Its Implementation. LNCS, vol. 922. Springer, Heidelberg (1995)
18. Dotti, F.L., Ribeiro, L., dos Santos, O.M., Pasini, F.: Verifying object-based graph grammars. Software and System Modeling 5(3), 289–311 (2006)
19. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. J. Comput. Syst. Sci. 64(2), 249–283 (2002)
20. Ehrig, H.: Introduction to COMPUGRAPH. In: [14], pp. 89–100
21. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. Fundam. Inform. 74(1), 31–61 (2006)

22. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
23. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. II. World Scientific, Singapore (1999)
24. Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.): ICGT 2004. LNCS, vol. 3256. Springer, Heidelberg (2004)
25. Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.): ICGT 2008. LNCS, vol. 5214. Springer, Heidelberg (2008)
26. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallelism, and Distribution, vol. III. World Scientific, Singapore (1999)
27. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: 14th Annual Symposium on Switching and Automata Theory, pp. 167–180. IEEE, Los Alamitos (1973)
28. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML activities using dynamic meta modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 76–90. Springer, Heidelberg (2007)
29. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: language and environment. In: [23], http://tfs.cs.tu-berlin.de/agg/
30. The FUJABA toolsuite (2006), http://www.fujaba.de
31. GBT — graph backwards tool, http://www.it.uu.se/research/group/mobility/adhoc/gbt
32. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: Grgen: A fast spo-based graph rewriting tool. In: [13], pp. 383–397
33. GETGRATS: General theory of graph transformation systems (1997), http://www.di.unipi.it/~andrea/GETGRATS/
34. Gorp, P.V., Keller, A., Janssens, D.: Transformation language integration based on profiles and higher order transformations. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 208–226. Springer, Heidelberg (2009)
35. Hausmann, J.H.: Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD thesis, University of Paderborn (2005)
36. Hausmann, J.H., Heckel, R., Sauer, S.: Towards dynamic meta modeling of UML extensions: An extensible semantics for UML sequence diagrams. In: Human-Centric Computing Languages and Environments (HCC), pp. 80–87. IEEE Computer Society, Los Alamitos (2001)
37. Holt, R.C., Schürr, A., Sim, S.E., Winter, A.: GXL: A graph-based standard exchange format for reengineering. Sci. Comput. Program. 60(2), 149–170 (2006)
38. Holzmann, G.: The SPIN Model Checker — Primer and Reference Manual. Addison-Wesley, Reading (2004)
39. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Karsten Ehrig, H.G. (ed.) Graph Transformation and Visual Modeling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 6 (2007)
40. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. Theor. Comput. Sci. 404(3), 256–274 (2008)
41. Kastenberg, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)
42. Kleppe, A.G., Rensink, A.: On a graph-based semantics for UML class and object diagrams. In: Ermel, C., Lara, J.D., Heckel, R. (eds.) Graph Transformation and Visual Modelling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 10. EASST (2008)

43. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 197–211. Springer, Heidelberg (2006)
44. König, B., Kozioura, V.: Augur 2 — a new version of a tool for the analysis of graph transformation systems. In: Bruni, R., Varró, D. (eds.) Graph Transformation and Visual Modeling Techniques (GT-VMT 2006). Electronic Notes in Theoretical Computer Science, vol. 211, pp. 201–210 (2008)
45. Kuske, S., Gogolla, M., Kreowski, H.J., Ziemann, P.: Towards an integrated graph-based semantics for UML. Software and System Modeling 8(3), 403–422 (2009)
46. Lambers, L.: A new version of GTXL: An exchange format for graph transformation systems. In: Proceedings of the International Workshop on Graph-Based Tools (GraBaTs). Electronic Notes in Theoretical Computer Science, vol. 127, pp. 51–63 (March 2005)
47. Lengyel, L., Levendovszky, T., Vajk, T., Charaf, H.: Realizing qvt with graph rewriting-based model transformation. In: Karsai, G., Taentzer, G. (eds.) Graph and Model Transformation (GraMoT). Electronic Communications of the EASST, vol. 4 (2006)
48. Levendovszky, T., Rensink, A., Van Gorp, P.: 5th International Workshop on Graph-Based Tools: The contest (grabats) (2009), http://is.ieis.tue.nl/staff/pvgorp/events/grabats2009
49. McMillan, K.L.: Methods for exploiting sat solvers in unbounded model checking. In: Formal Methods and Models for Co-Design (MEMOCODE), p. 135. IEEE Computer Society, Los Alamitos (2003)
50. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Englewood Cliffs (1989)
51. Milner, R.: Bigraphical reactive systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 16–35. Springer, Heidelberg (2001)
52. Milner, R.: Pure bigraphs: Structure and dynamics. Inf. Comput. 204(1), 60–122 (2006)
53. Nagl, M.: Set theoretic approaches to graph grammars. In: Ehrig, H., Nagl, M., Rosenfeld, A., Rozenberg, G. (eds.) Graph Grammars 1986. LNCS, vol. 291, pp. 41–54. Springer, Heidelberg (1987)
54. OMG: Meta object facility (MOF) core specification, v2.0. Document formal/06-01-01, Object Management Group (2006), http://www.omg.org/cgi-bin/doc?formal/06-01-01
55. OMG: Unified modeling language, superstructure, v2.2. Document formal/09-02-02, Object Management Group (2009), http://www.omg.org/cgi-bin/doc?formal/09-02-02
56. Palacz, W.: Algebraic hierarchical graph transformation. J. Comput. Syst. Sci. 68(3), 497–520 (2004)
57. Pennemann, K.H.: Development of Correct Graph Transformation Systems. PhD thesis, University of Oldenburg, Oldenburg (2009), http://oops.uni-oldenburg.de/volltexte/2009/948/
58. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
59. Rensink, A.: Isomorphism checking in GROOVE. In: Zündorf, A., Varró, D. (eds.) Graph-Based Tools (GraBaTs). Electronic Communications of the EASST, European Association of Software Science and Technology, vol. 1 (September 2007)
60. Rensink, A.: Compositionality in graph transformation. In: Abramsky, S., et al. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 309–320. Springer, Heidelberg (2010)
61. Rensink, A., Distefano, D.S.: Abstract graph transformation. In: Mukhopadhyay, S., Roychoudhury, A., Yang, Z. (eds.) Software Verification and Validation, Manchester. Electronic Notes in Theoretical Computer Science, vol. 157, pp. 39–59 (May 2006)
62. Rensink, A., Taentzer, G.: Agtive 2007 graph transformation tool contest. In: [71], pp. 487–492, http://www.informatik.uni-marburg.de/~swt/agtive-contest

63. Rensink, A., Van Gorp, P.: Graph transformation tool contest 2008. Software Tools for Technology Transfer (2009) Special section; in preparation, http://fots.ua.ac.be/events/grabats2008
64. Rensink, A., Zambon, E.: A type graph model for java programs. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 237–242. Springer, Heidelberg (2009)
65. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific, Singapore (1997)
66. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
67. Saksena, M., Wibling, O., Jonsson, B.: Graph grammar modeling and verification of ad hoc routing protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
68. Schmidt, Á., Varró, D.: Checkvml: A tool for model checking visual modeling languages. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 92–95. Springer, Heidelberg (2003)
69. Schürr, A.: Programmed graph replacement systems. In: [65], pp. 479–546
70. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: [25], pp. 411–425
71. Schürr, A., Nagl, M., Zündorf, A. (eds.): Applications of Graph Transformations with Industrial Relevance (AGTIVE). LNCS, vol. 5088. Springer, Heidelberg (2008)
72. SEGRAVIS: Syntactic and semantics integration of visual modelling techniques (2002), http://www.segravis.org/
73. SENSORIA: Software engineering for service-oriented overlay computers (2005), http://www.sensoria-ist.eu
74. Sleep, M.R.: SEMAGRAPH: The theory and practice of term graph rewriting. In: [14], pp. 268–276
75. Taentzer, G.: Parallel high-level replacement systems. Theor. Comput. Sci. 186(1-2), 43–81 (1997)
76. Taentzer, G.: Distributed graphs and graph transformation. Applied Categorical Structures 7(4) (1999)
77. Taentzer, G.: Towards common exchange formats for graphs and graph transformation systems. In: Uniform Approaches to Graphical Process Specification Techniques (UNIGRA). Electronic Notes in Theoretical Computer Science, vol. 44, pp. 28–40 (2001)
78. Taentzer, G., Biermann, E., Bisztray, D., Bohnet, B., Boneva, I., Boronat, A., Geiger, L., Geiß, R., Horvath, Á., Kniemeyer, O., Mens, T., Ness, B., Plump, D., Vajk, T.: Generation of Sierpinski triangles: A case study for graph transformation tools. In: [71], pp. 514–539
79. Taentzer, G., Fischer, I., Koch, M., Volle, V.: Distributed graph transformation with application to visual design of distributed systems. In: [26], ch. 6, pp. 269–340
80. Taentzer, G., Rensink, A.: Ensuring structural constraints in graph-based models with type inheritance. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 64–79. Springer, Heidelberg (2005)

Graph Transformation by Computational Category Theory

Mark Minas[1] and Hans Jürgen Schneider[2]

[1] Universität der Bundeswehr München, Germany
Mark.Minas@unibw.de
[2] Universität Erlangen, Germany
schneider@informatik.uni-erlangen.de

Abstract. The categorical approach is well-suited for concise definitions of graph transformation concepts. At the same time, it allows for elegant proofs of their properties. We show that these categorical concepts also allow for a similarly simple and modular implementation of graphs and graph transformations by following Rydeheard and Burstall's idea of Computational Category Theory. We first present an implementation of some categorical definitions and constructions, e.g., colimits, in Java, and we demonstrate how this language supports the genericity of the categorical approach. We then show that applying the constructions to the category of sets as well as the category of graphs already provides an implementation of graph transformations that can be used as the foundation of an extensible graph transformation language.

1 Introduction

Graphs are the most frequently used complex data structure in computer science. Typical examples are simple list structures, trees, Petri nets, and even the visual languages of the UML. These data structures are usually not fixed, but are continuously modified and transformed. Graph transformation is the field of computer science that examines such transformations of graphs in a more general setting; it considers systems that are described by graphs that can be transformed by a selected set of transformation rules only. Such graph transformation systems are used as abstract models of concrete systems (e.g., programming and software development systems [15], compilers, visual editors [14]). Using graph transformation systems as abstract models has several benefits: Graph transformation is a powerful modeling approach, it can be effectively used to model non-trivial systems. Graph transformation systems can thus be used for defining the semantics of the concrete system or for proving its properties. On the other hand, the modeled systems can be implemented – at least in a prototypical way – if there is an implementation of the graph transformation system. This was and is the main motivation for many tools that are based on graph transformations, e.g., PROGRES [21], AGG[9], DIAGEN [14] to mention just a few. However, each of those tools is based on a specific type of graphs (e.g., directed node labeled graphs or hypergraphs) which was selected during design of the tool; the tool cannot be applied to different kinds of graphs.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 33–58, 2010.

This paper describes a different way to implement graph transformations. Instead of selecting a specific version of graphs, we have selected the categorical approach to graph transformations [4,6,7,19], which is highly generic: All the proofs and constructions are valid for various types of graphs (unlabeled, node labeled, edge labeled, different ways of labeling, etc.). Since modern programming languages like Java support generic concepts, it is a promising idea to present the categorical approach to graph transformations in such a language. We can implement the general definitions and constructions without referring to special versions of graphs. If we subsequently consider a special type of graphs, we have to define only some basic operations in detail. All the other stuff is inherited from the generic modules, i.e., it must be implemented only once. This allows for rapidly prototyping new graph transformation approaches. Closeness of categorical constructions and their implementation makes the presented approach well suited for education purposes, too.

A related concept for implementing concepts of category theory, but without application to graph transformation, has been published by Rydeheard and Burstall using ML [18]. Their approach is mainly based on polymorphism, whereas we can take advantage of the object-oriented paradigm which allows for a more modular implementation.

The following section introduces the mathematical concept of *categories* with *objects* and *morphisms*. A very general representation of those concepts in Java is presented and demonstrated using the example of $\mathcal{S}et$, the category of finite sets and total functions on them. Section 3 introduces colimits and their general realization in Java, again demonstrated using the example of $\mathcal{S}et$. Colimits are of particular interest for the categorical approach to graph transformations because it is based on *pushouts*, a specific kind of colimit. Sections 4 and 5 represent unlabeled and labeled graphs and their categories that are then used in Section 6 to realize the categorical approach to graph transformations for unlabeled and labeled graphs following the single-pushout approach. Section 7 concludes the paper.

Please note that we assume that the reader is familiar with basic category theory, Java, and Java generics in particular.

2 Categories

A category is a very general concept in mathematics; each category consists of a class of *objects* and a class of *morphisms* between objects. The class of morphisms is closed under composition, and the existence of identities and the law of associativity for the composition operation are assumed. We use the definition given by Herrlich and Strecker [10]:

Definition 2.1. *A category is a quintuple* $\mathcal{C} = (\mathrm{Obj}_C, \mathrm{Mor}_C, \mathrm{dom}_C, \mathrm{codom}_C, \cdot_C)$ *with the total functions* $\mathrm{dom}_C, \mathrm{codom}_C : \mathrm{Mor}_C \to \mathrm{Obj}_C$ *and the partial function* $\cdot_C : \mathrm{Mor}_C \times \mathrm{Mor}_C \to \mathrm{Mor}_C$ *with the following properties:*

- $g \cdot_C f$ *exists if and only if* $\mathrm{codom}_C(f) = \mathrm{dom}_C(g)$
- $\mathrm{dom}_C(g \cdot_C f) = \mathrm{dom}_C(f)$ *and* $\mathrm{codom}_C(g \cdot_C f) = \mathrm{codom}_C(g)$

```
public interface Cat<O, M extends Mor<O, M>> {
    M id(O obj);
}

public interface Mor<O, M> {
    O dom();
    O codom();
    M comp(M o);
}
```

Fig. 1. Basic type definitions for implementing categories and morphisms

- *Associativity:* $(h \cdot_C g) \cdot_C f = h \cdot_C (g \cdot_C f)$ *for all morphisms* $f, g, h \in \mathrm{Mor}_C$ *with* $\mathrm{codom}_C(f) = \mathrm{dom}_C(g)$ *and* $\mathrm{codom}_C(g) = \mathrm{dom}_C(h)$.
- *Identities: For each object* $A \in \mathrm{Obj}_C$*, there exists a morphism* $\mathrm{id}_A \in \mathrm{Mor}_C$ *such that* $\mathrm{id}_B \cdot_C f = f = f \cdot_C \mathrm{id}_A$ *for each morphism* $f \in \mathrm{Mor}_C$ *and* $A = \mathrm{dom}(f)$ *and* $B = \mathrm{codom}(f)$.

In the following, we write $f : A \to B$ for a morphism $f \in \mathrm{Obj}_C$ and objects $A = \mathrm{dom}_C(f), B = \mathrm{codom}_C(f)$.

An object-oriented representation of a category with its objects and morphisms requires three collaborating classes, one class for the category, one for its objects, and one for its morphisms. Fig. 1 shows the fundamental Java types that provide a basis for such implementations: The generic Java interface[1] Cat<O,M> specifies a class representing a category; its type parameters O and M represent the corresponding object and morphism types. M is a bounded type parameter with Mor<O,M> as an upper bound, i.e., each morphism class must be derived from the generic interface Mor<O,M> which uses the same type parameters. This interface provides methods dom() and codom() for the corresponding functions in Def. 2.1. Moreover, it specifies method M comp(M mor). The concrete morphism class must provide the implementation of the composition operation for this method such that calling m1.comp(m2) computes the composition $m_1 \cdot m_2$. The return type which is also the type parameter M of Cat<O,M> makes sure that the types of the composed morphisms and the computed morphisms match. The concrete category class implementing Cat<O,M>, finally, must provide an implementation of the method M id(O obj) that computes the identity morphism for each passed Java object representing a categorical object. Note that the type parameter O is not bounded; this allows to use any Java type for representing objects. Only the choice of types for categories and morphisms is restricted by the presented Java types Cat<O,M> and Mor<O,M>.

As an example, we implement the well-known category $\mathcal{S}et$, which has all finite sets as objects and all total functions from finite sets to finite sets as morphisms. A morphism $f : A \to B$, hence, is a total function $f : A \to B$. Java already provides several set implementations in its API; all of them implement the interface

[1] For a description of generic Java data types, see, e.g., [2,22].

```
public class SetCat implements Cat<Set<?>, SetMor> {
    public static final SetCat SET = new SetCat();
    private SetCat() {}

    public SetMor id(Set<?> o) {
        SetMor mor = new SetMor(o, o);
        for (Object elem : o) mor.map(elem, elem);
        return mor;
    }
}

public class SetMor implements Mor<Set<?>, SetMor> {
    private final Map<Object, Object> map = new HashMap<Object, Object>();
    private final Set<?> from, to;
    public SetMor(Set<?> from, Set<?> to) { this.from = from; this.to = to; }

    public Object applyTo(Object o) { return map.get(o); }
    public void map(Object f, Object t) { map.put(f, t); }
    public Set<?> codom() { return to; }
    public Set<?> dom() { return from; }
    public SetMor comp(SetMor m) {
        assert dom().equals(m.codom());
        SetMor mor = new SetMor(m.dom(), codom());
        for (Object o : m.dom()) mor.map(o, applyTo(m.applyTo(o)));
        return mor;
    }
}
```

Fig. 2. Abstract implementation of category $\mathcal{S}et$ and its morphisms

Set<E> where E represents the type of all set elements.[2] The type Set<?> with unknown element type, therefore, is an obvious choice for representing objects of category $\mathcal{S}et$. The wildcard ? allows to have finite sets of arbitrary Java objects. This definition allows to use instances of any class implementing Set<E> as $\mathcal{S}et$ objects.

Fig. 2 shows the corresponding classes representing category $\mathcal{S}et$ and its morphisms. Class SetMor implements the morphism interface Mor<O,M>; the object type parameter O is bound to the type Set<?> of all $\mathcal{S}et$ objects whereas the morphism type parameter is bound to class SetMor itself. The domain and codomain sets of a total function are passed as constructor parameters. Member attribute map contains the (initially empty) value table of the function which gets populated using the map method. The applyTo method allows for value table lookup. Note that method SetMor comp(SetMor m) implements method M comp(M mor). The assertion makes sure that only functions with appropriate domain and

[2] Interface Set<E> has actually the fully qualified name java.util.Set<E>. We omit all package names in this paper for the sake of simplicity.

codomain, respectively, get composed.[3] The composition of two functions is computed in the obvious way by iterating over all elements of the domain set and computing the resulting function values in the usual way.[4]

Class SetCat implements interface Cat<O,M> with the type parameters being bound in the same way as for SetMor. SetCat is realized as a singleton class since there is only one category *Set*; the only SetCat instance can be accessed through class variable SET. Method SetMor id(Set<?> o) provides an implementation of method M id(O o) of interface Cat<O,M> and computes the identity function for the set being passed as parameter.

Class SetCat as shown in Fig. 2 is going to be extended in the next section where we use category *Set* again as an example for demonstrating the realization of colimit constructions.

The presented implementation of category *Set* is rather straightforward, but not very efficient. Every function application, e.g., requires a hash map lookup. A more efficient solution could represent sets by simple arrays that hold the set elements. Elements would be mapped to elements of another set by mapping their positions in the corresponding arrays. This mapping could again be represented by an array, i.e., function application would be more efficient as long as the index position of an element is known. However, we have refrained from presenting this solution here since it is more technical and less straightforward than the presented one.

3 Colimits

An advantage of category theory is that it provides us with general constructions applicable in different areas, e.g., we can construct all finite colimits in any category that has initial object, coproduct, and coequalizer, and this construction does not depend on the special category. We implement the colimits as interfaces as shown in Fig. 3. We can define the interfaces by simply looking at the corresponding definitions, e.g., [18]: The *initial object* of a category, if it exists, is an object such that there is a unique morphism to any object of the category. This requirement of the existence of a unique morphism is the so-called *universal property* of the initial object. We represent initial objects by the generic interface InitialObj<O,M> where the type parameters O and M again represent the types used for objects and morphisms, respectively. The actual initial object can be retrieved by method O obj(). The universal property is represented by a method, too. Method M univ(O o) returns the unique morphism to an object when this object is passed as parameter o. Hence, implementations of this interface must provide a constructive proof of the existence of the unique morphism. The

[3] Trying to compose two inappropriate functions means a programming error. The Java runtime environment then throws an exception. Assertion checking can be switched off completely when starting the Java environment. Time-consuming checks are then avoided, but errors may get missed that way.

[4] We have omitted implementations of the equals and hashCode methods which are actually required later.

```
public interface InitialObj<O, M> {
    O obj();
    M univ(O o);
}

public interface Coproduct<O, M> {
    O obj();
    M mor1();
    M mor2();
    M univ(M m1, M m2);
}
```

```
public interface Coequalizer<O, M> {
    O obj();
    M mor();
    M univ(M m);
}

public interface Pushout<O, M> {
    O obj();
    M mor1();
    M mor2();
    M univ(M m1, M m2);
}
```

Fig. 3. Abstract representation of some colimits

```
public interface CatWithInitialObj<O, M extends Mor<O, M>>
                 extends Cat<O, M> {
    InitialObj<O, M> initialObj();
}

public interface CatWithCoproduct<O, M extends Mor<O, M>>
                 extends Cat<O, M> {
    Coproduct<O, M> coproduct(O o1, O o2);
}

public interface CatWithCoequalizer<O, M extends Mor<O, M>>
                 extends Cat<O, M> {
    Coequalizer<O, M> coequalizer(M g, M h);
}

public interface CatWithPushout<O, M extends Mor<O, M>>
                  extends Cat<O, M> {
    Pushout<O, M> pushout(M f, M g);
}

public interface CatWithColimits<O, M extends Mor<O, M>>
                 extends CatWithInitialObj<O, M>, CatWithCoproduct<O, M>,
                         CatWithCoequalizer<O, M>, CatWithPushout<O, M> {}
```

Fig. 4. Representation of categories that have specific colimits

implementation of a category that has an initial object must implement interface CatWithInitialObj<O,M> (see Fig. 4). Its method initialObj() computes the initial object of the category, which on its part, contains the method to compute the universal property.

Other colimits are (binary) coproducts, coequalizers, and pushouts. The (binary) *coproduct* of two objects $o_1, o_2 \in \text{Obj}$ is an object $o \in \text{Obj}$ with two

morphisms $m_1 : o_1 \to o$ and $m_2 : o_2 \to o$ such that for any object $o' \in \text{Obj}$ and morphisms $m'_1 : o_1 \to o'$ and $m'_2 : o_2 \to o'$, there is a unique morphism $u : o \to o'$ such that $m'_1 = u \cdot m_1$ and $m'_2 = u \cdot m_2$. The *coequalizer* of two morphisms $f, g : o_1 \to o_2$ with common domain $o_1 \in \text{Obj}$ and common codomain $o_2 \in \text{Obj}$ is an object $o \in \text{Obj}$ together with a morphism $m : o_2 \to o$ such that $m \cdot f = m \cdot g$ and for any object $o' \in \text{Obj}$ and morphism $m' : o_2 \to o'$ with $m' \cdot f = m' \cdot g$, there is a unique morphism $u : o \to o'$ such that $m' = u \cdot m$. Finally, the *pushout* of two morphisms $f : o_0 \to o_1$ and $g : o_0 \to o_2$ with common domain $o_0 \in \text{Obj}$ consists of an object $o \in \text{Obj}$ together with two morphisms $m_1 : o_1 \to o$ and $m_2 : o_2 \to o$ such that $m_1 \cdot f = m_2 \cdot g$ and for any object $o' \in \text{Obj}$ and morphisms $m'_1 : o_1 \to o'$ as well as $m'_2 : o_2 \to o'$ such that $m'_1 \cdot f = m'_2 \cdot g$ there is a unique morphism $u : o \to o'$ such that $m'_1 = u \cdot m_1$ and $m'_2 = u \cdot m_2$. We say that (m_1, m_2) is the pushout of (f, g).

Similar to initial objects, we represent coproducts, coequalizers, and pushouts by generic interfaces as shown in Fig. 3. They provide methods for retrieving the object of the colimit as well as the corresponding morphisms. Again, the universal properties, i.e., the existence of a unique morphism with the specified property, can be calculated using a method univ(...) that receives as parameters the morphisms that the unique morphism is calculated for. The implementation of a category that has coproducts, coequalizers, or pushouts must implement the corresponding interfaces as shown in Fig. 4. For instance, the implementation of a category with coequalizers must implement the method coequalizer(M g, M h) that computes the coequalizer for the morphisms that are passed as arguments g and h. We provide the generic interface CatWithColimits<O, M> as a convenience interface for implementations of categories that have all four kinds of colimits. Actually, the existence of initial objects and pushouts is sufficient for the existence of any finite colimit in this category [18]. Similarly, the existence of pushouts follows from the existence of (binary) coproducts and coequalizers. We will use this fact below to compute pushouts based on coproduct and coequalizer calculations.

Set is an example category that has all colimits. The implementing class SetCat (Fig. 2), therefore, must implement CatWithColimits<Set<?>, SetMor> (which includes interface Cat<Set<?>, SetMor>, too) and the additional methods required by CatWithColimits<Set<?>, SetMor>. Fig. 5 shows their implementation for *Set* as described below:

The initial object of *Set* is the empty set, and the unique morphism to any object is the empty function. The empty set can be represented by Collections.EMPTY_SET of the Java API; the empty function is created by just instantiating SetMor with the corresponding domain and codomain.

The (binary) coproduct of two sets o_1 and o_2 is their disjoint union together with their natural injections. Fig. 6 shows this implementation. The constructor populates member variable union with the disjoint union of the sets being passed as parameters. Each element of set1 and set2 is wrapped in a Pair object (Class Pair is simply a data class that is not shown here). The second component of the

```
public InitialObj<Set<?>, SetMor> initialObj() {
    return new InitialObj<Set<?>, SetMor>() {
        public Set<?> obj() { return EMPTY_SET; }
        public SetMor univ(Set<?> o) { return new SetMor(obj(), o); }
    };
}

public Coproduct<Set<?>, SetMor> coproduct(Set<?> o1, Set<?> o2) {
    return new DisjointUnion(o1, o2);
}

public Coequalizer<Set<?>, SetMor> coequalizer(SetMor f, SetMor g) {
    assert f.dom().equals(g.dom()) && f.codom().equals(g.codom());
    Partition partition = new Partition(f.codom());
    for (Object x : f.dom())
        partition.sameSubset(f.applyTo(x), g.applyTo(x));
    return partition;
}
```

Fig. 5. Computing initial objects, binary coproducts, and coequalizers in class SetCat

pair allows to distinguish elements of the one set from elements of the other.[5] The universal property, finally, is computed by method univ(SetMor n1, SetMor n2). If $m_1 : o_1 \to o$ and $m_2 : o_2 \to o$ are the natural injections of the coproduct, and $n_1 : o_1 \to o'$ and $n_2 : o_2 \to o'$ two morphisms, it has to compute a morphism u such that $n_1 = u \cdot m_1$ and $n_2 = u \cdot m_2$. It is easy to see that $u : o \to o'$ must be defined as follows:

$$u(e) = \begin{cases} n_1(x) & \text{if } m_1(x) = e \\ n_2(x) & \text{if } m_2(x) = e \end{cases}$$

which is realized by the univ method. Note the assertions in this method that make sure that morphisms n_1 and n_2 have appropriate domains and codomains.

The coequalizer of two functions $f, g : a \to b$ with common domain a and codomain b (note the assertions in Fig. 5) is a set of equivalence classes of b. These equivalence classes are defined by the finest equivalence relation containing the relation $\sim$ on b with $f(x) \sim g(x)$ for any $x \in a$. Determining this equivalence relation is the well-known union-find problem [13] that is implemented by class Partition. The equivalence relation is built up by iterating over all elements x of $\mathrm{dom}(f)$ and building the union of the subsets of $f(x)$ and $g(x)$. Class Partition is not presented here, but its implementation is straightforward [3,13].

The methods for calculating initial objects, (binary) coproducts, and coequalizers must be implemented for every category that has all finite colimits. However, pushouts need not be implemented anew since there is a canonical construction of pushouts that makes use of coproducts and coequalizers [18] as shown in Fig. 7: When constructing the pushout of two morphisms $f : o \to o_1$ and $g : o \to o_2$, we first build the coproduct cp of o_1 and o_2 and then the coequalizer

[5] Using numbers is an approach that can easily be extended to n-ary coproducts.

```
class DisjointUnion implements Coproduct<Set<?>, SetMor> {
    private final SetMor m1, m2;
    private final Set<Pair> union;

    public DisjointUnion(Set<?> set1, Set<?> set2) {
        union = new HashSet<Pair>();
        m1 = new SetMor(set1, union);
        m2 = new SetMor(set2, union);
        for (Object o : set1) {
            Pair p = new Pair(o, "1");
            union.add(p);
            m1.map(o, p);
        }
        for (Object o : set2) {
            Pair p = new Pair(o, "2");
            union.add(p);
            m2.map(o, p);
        }
    }

    public Set<?> obj() { return union; }
    public SetMor mor1() { return m1;}
    public SetMor mor2() { return m2; }

    public SetMor univ(SetMor n1, SetMor n2) {
        assert n1.codom().equals(n2.codom());
        assert n1.dom().equals(m1.dom()) && n2.dom().equals(m2.dom());
        SetMor u = new SetMor(union, n1.codom());
        for (Object o : n1.dom())
            u.map(m1.applyTo(o), n1.applyTo(o));
        for (Object o : n2.dom())
            u.map(m2.applyTo(o), n2.applyTo(o));
        return u;
    }
}
```

Fig. 6. Computing the binary coproduct in $\mathcal{S}et$ by the disjoint union

ce of the morphisms $m_1 \cdot f$ and $m_2 \cdot g$ where m_1 and m_2 are the coproduct morphisms. The pushout morphisms are $mor_1 = m \cdot m_1$ and $mor_2 = m \cdot m_2$ where m is the coequalizer morphism. The universal property can be constructed from the universal properties of the coproduct and the coequalizer: Given two morphisms $n_1 : o_1 \to o'$ and $n_2 : o_2 \to o'$ such that $n_1 \cdot f = n_2 \cdot g$, one can find the unique morphism $u_1 : cp \to o'$ such that $u_1 \cdot m_1 = n_1$ and $u_1 \cdot m_2 = n_2$ by the coproduct's universal property, hence $u_1 \cdot (m_1 \cdot f) = u_1 \cdot (m_2 \cdot g)$. Therefore, we can find a unique morphism $u_2 : ce \to o'$ such that $u_2 \cdot m = u_1$ by the coequalizer's universal property. It is easy to see that u_2 satisfies the conditions of a pushout's universal property [19].

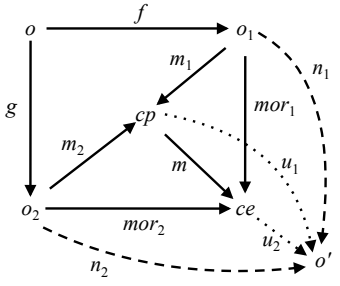


Fig. 7. Pushout construction from (binary) coproduct and coequalizer

```
public class CPCEPushout<O, M extends Mor<O, M>,
                         C extends CatWithCoproduct<O, M> &
                                   CatWithCoequalizer<O, M>>
        implements Pushout<O, M> {
    private final Coproduct<O, M> cp;
    private final Coequalizer<O, M> ce;
    private final M m1, m2;

    public CPCEPushout(C cat, M f, M g) {
        assert f.dom().equals(g.dom());
        cp = cat.coproduct(f.codom(), g.codom());
        ce = cat.coequalizer(cp.mor1().comp(f), cp.mor2().comp(g));
        m1 = ce.mor().comp(cp.mor1());
        m2 = ce.mor().comp(cp.mor2());
    }

    public O obj() { return ce.obj(); }
    public M mor1() { return m1; }
    public M mor2() { return m2; }
    public M univ(M n1, M n2) { return ce.univ(cp.univ(n1, n2)); }
}
```

Fig. 8. Computing the pushout in a category that has coproducts and coequalizers

A generic implementation of this canonical construction of pushouts is shown in Fig. 8. The presented class implements the Pushout<O,M> interface. The existence of coproducts and coequalizers and, hence, the corresponding methods for constructing them, is enforced by the bounded type parameter C with upper bounds CatWithCoproduct<O,M> as well as CatWithCoequalizer<O,M>. An instance of a corresponding Java class representing the category in use is passed to the constructor; its methods coproduct(...) and coequalizer(...) are used for constructing coproduct, coequalizer, and the universal property of the pushout as described above.

```
public abstract class CocoCat<O, M extends Mor<O, M>>
                        implements CatWithColimits<O, M> {
    public Pushout<O, M> pushout(M f, M g) {
        return new CPCEPushout<O, M, CatWithColimits<O, M>>(this, f, g);
    }
}
```

Fig. 9. Implementation of a category with pushouts based on the canonical construction shown in Fig. 8

We can make use of the canonical construction of pushouts in any class that has all binary coproducts and coequalizers. Fig. 9 shows the abstract class CocoCat<O,M>[6] that has all finite colimits as described in Fig. 4. However, the pushout functionality is already implemented based on the canonical construction using coproducts and coequalizers. Therefore, concrete subclasses of this class already have the pushout(...) method available without being forced to implement it. Class SetCat (Fig. 2) can make use of it: instead of implementing Cat<Set<?>, SetMor> as shown in Fig. 2, it can extend CocoCat<Set<?>, SetMor> and, together with the colimit implementations shown in Fig. 5, can then be used to compute pushouts in $\mathcal{S}et$.

4 Graphs

This and the following sections consider unlabeled and labeled graphs. This section starts with unlabeled graphs that are extended by labeling in the following section. Formally, a directed graph G is a quadruple $G = (N_G, E_G, s_G, t_G)$ where N_G is a finite set of nodes, E_G a finite sets of edges, and $s_G, t_G : E_G \to N_G$ are total functions assigning a source and a target node to each edge. A *graph morphism* $m : G \to H$ is a pair $m = (m_\mathrm{N}, m_\mathrm{E})$ of total functions $m_\mathrm{N} : N_G \to N_H$ and $m_\mathrm{E} : E_G \to E_H$ that map nodes and edges, respectively, and preserve the structure of graphs, i.e., $s_H \cdot m_\mathrm{E} = m_\mathrm{N} \cdot s_G$ and $t_H \cdot m_\mathrm{E} = m_\mathrm{N} \cdot t_G$.

Following these definitions, it is straightforward to represent graphs and graph morphisms by Java interfaces as shown in Fig. 10. Finite sets and total functions on finite sets are represented by type parameters S and F, respectively. These parameters can be bound, e.g., to Set<?> and SetMor as defined in the last sections.

Directed graphs make up a category $\mathcal{G}raph$ with directed graphs as objects and graph morphisms as morphisms. $\mathcal{G}raph$ is based on $\mathcal{S}et$ since each graph consists of two finite sets as well as two total functions on these sets, and each graph morphism consists of two total functions on finite sets, too. We have used

[6] The name *CocoCat* stands for *cocomplete category* which is a category having all colimits. A category having initial object, all binary coproducts, and all coequalizers actually has all finite colimits. However, the implementation of class CocoCat<O,M> in Fig. 9 does not show the general construction of all finite colimits.

```
public interface Graph<S, F> {
    S nodes();
    S edges();
    F source();
    F target();
}
```

```
public interface GraphMor<F> {
    F nodeMor();
    F edgeMor();
}
```

Fig. 10. Representing graphs and graph morphisms

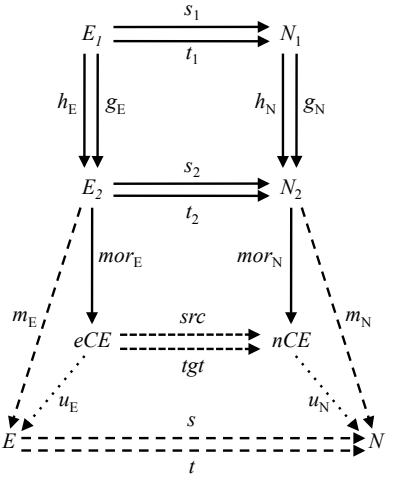


Fig. 11. Constructing coequalizers in $\mathcal{G}raph$

this fact in the definition of the corresponding interfaces in Fig. 10. $\mathcal{G}raph$, actually, "inherits" the property of having all colimits from $\mathcal{S}et$; colimits in $\mathcal{G}raph$ can be constructed by creating the corresponding colimits for the node and edge sets separately, and then using the universal properties of colimits in $\mathcal{S}et$ for constructing source and target functions as well as graph morphisms [18]. We demonstrate this construction for $\mathcal{G}raph$ coequalizers. We will then use this and similar constructions for other colimits when implementing $\mathcal{G}raph$ with its colimits.

Fig. 11 shows two graphs $G_1 = (N_1, E_1, s_1, t_1)$ and $G_2 = (N_2, E_2, s_2, t_2)$ with graph morphisms $g, h : G_1 \to G_2$. The sets nCE and eCE together with the functions $mor_N : N_2 \to nCE$ and $mor_E : E_2 \to eCE$ are the $\mathcal{S}et$ coequalizers of the function pairs $g_N, h_N : N_1 \to N_2$ and $g_E, h_E : E_1 \to E_2$, respectively. These coequalizers are simply called nCE and eCE in the following. $G_{CE} = (nCE, eCE, src, tgt)$ together with the graph morphism $mor = (mor_N, mor_E)$ is the $\mathcal{G}raph$ coequalizer of g and h. The universal property of coequalizer eCE allows for computing functions src and tgt for $mor_N \cdot s_2$ and $mor_N \cdot t_2$, respectively. Given a graph $G = (N, E, s, t)$ and a graph morphism $m : G_2 \to G$ with $m \cdot g = m \cdot h$, we compute the universal property of $\mathcal{G}raph$ coequalizer G_{CE}, i.e., the unique graph morphism $u : G_{CE} \to G$ by constructing the pair (u_N, u_E) of functions using the universal properties of nCE and eCE.

```
public abstract class GraphCat<SO, SM extends Mor<SO, SM>,
                               GO extends Graph<SO, SM>,
                               GM extends GraphMor<SM> & Mor<GO, GM>>
                      extends CocoCat<GO, GM> {
    private final CatWithColimits<SO, SM> setCat;
    public GraphCat(CatWithColimits<SO, SM> setCat) { this.setCat = setCat; }

    public abstract GM makeMor(GO from, GO to, SM nodeMor, SM edgeMor);
    protected abstract GO initialObj(InitialObj<SO, SM> cIO);
    protected abstract GO coproduct(GO o1, GO o2,
                                    BinCoproduct<SO, SM> nCP,
                                    BinCoproduct<SO, SM> eCP);
    protected abstract GO coequalizer(GM g, GM h,
                                      Coequalizer<SO, SM> nCE,
                                      Coequalizer<SO, SM> eCE);
    public GM id(GO o) {
        SM nm = setCat.id(o.nodes());
        SM em = setCat.id(o.edges());
        return makeMor(o, o, nm, em);
    }
    public InitialObj<GO, GM> initialObj() {
        return new GraphInitialObj();
    }
    public BinCoproduct<GO, GM> coproduct(GO o1, GO o2) {
        return new GraphBinCoproduct(o1, o2);
    }
    public Coequalizer<GO, GM> coequalizer(GM g, GM h) {
        return new GraphCoequalizer(g, h);
    }
    ...
}
```

Fig. 12. Implementation of category $\mathcal{G}raph$

All categorical constructions in $\mathcal{G}raph$ are similar to this coequalizer construction; all of them are based on $\mathcal{S}et$ and its colimits. This observation gives rise to the implementation of $\mathcal{G}raph$ by the generic class GraphCat<SO,SM,GO,GM> as shown in Fig. 12 as an abstract base-class for representing graph-based categories. This class is not only used for representing the category $\mathcal{G}raph$ of all unlabeled graphs, but also category $\mathcal{LG}raph$ of labeled graphs in the next section. The categorical constructions for computing colimits are encapsulated in inner classes GraphInitialObj, GraphBinCoproduct, and GraphCoequalizer, which are omitted in Fig. 12, indicated by the ellipsis "...". Class GraphCoequalizer is described later. Creation of colimit objects is specific for the concrete kind of graphs (e.g., unlabeled graphs or labeled graphs). Abstract methods initialObject(...), coproduct(...), and coequalizer(...) are provided for that purpose.

The type parameters SO and SM represent the object and morphism types of the chosen $\mathcal{S}et$ implementation. One can bind those parameters to Set<?> and

SetMor in order to use the *Set* implementation presented in the previous sections (see derived class UGraphCatImpl in Fig. 15). As already noted, this *Set* implementation has been selected for presentation reasons and is less efficient than other possible implementations. Providing type parameters SO and SM allows for using this *Graph* implementation with other, more efficient *Set* implementations without any modification.

GraphCat is also generic with respect to the actual implementations of graphs and graph morphisms that are represented by type parameters GO and GM. Upper bounds require to implement the interfaces shown in Fig. 10. The actual graph morphism class, however, must also implement Mor<GO,GM>, required by superclass CocoCat<GO,GM> (Fig. 9). Note that interface GraphMor<S,F> (Fig. 10) does not extend Mor<GO,GM>. It would have been possible to extend Mor<GO,GM> by GraphMor<S,F>, but that would have required to provide GraphMor with the same set of type parameters like GraphCat. Keeping GraphMor<S,F> separate from Mor<GO,GM> simplifies data types and reduces coupling. Concrete implementations simply have to implement both interfaces (e.g., see class GraphMorImpl in Fig. 15).

The colimit constructions make use of the chosen *Set* implementation. A GraphCat<SO,SM,GO,GM> instance must have access to an instance of the class implementing category *Set* that is passed as a constructor parameter and made available by method setCat(). Note that the type of this instance is CatWithColimits<SO,SM>, i.e., the implementation of any category with colimits could be used. It is the client's responsibility to use an implementation that goes with *Set*.

Each GraphCat<SO,SM,GO,GM> instance must be able to create new graph objects and morphisms as instances of the type being bound to GO and GM. Morphisms are created using factory method makeMor(...). This method is used, e.g., by method GM id(GO o) that creates the identity morphism of the specified graph by first computing the identity morphisms of the graph's node and edge sets, and then calling the morphism factory method.

In GraphCat<SO,SM,GO,GM>, there is no factory method for creating graph objects because it is used as base-class not only for representing the category *Graph* of unlabeled graphs, but also for labeled graphs and, possibly, other graphs, too. Graph objects require different methods for each different graph kind: Unlabeled graphs can be created by just passing a source and a target function; labeled graphs require additional parameters. Object factory methods, therefore, cannot belong to GraphCat<SO,SM,GO,GM>, but must be specified in subclasses representing specific categories. Fig. 13 shows subclass UGraphCat<SO,SM,GO,GM> for the category *Graph* of unlabeled graph and its object factory method makeObj(...). This method is abstract because the object type GO is generic in this class. A concrete implementation is shown later in Fig. 15.

Graph objects are created, e.g., during the construction of colimits. Implementations of these constructions are split into two parts: The general construction being independent of the concrete kind of graph-based category is implemented in inner classes GraphInitialObj, GraphBinCoproduct, and GraphCoequalizer

```
public abstract class UGraphCat<SO, SM extends Mor<SO, SM>,
                                GO extends Graph<SO, SM>,
                                GM extends GraphMor<SM> & Mor<GO, GM>>
                    extends GraphCat<SO, SM, GO, GM> {
    public UGraphCat(CatWithColimits<SO, SM> setCat) { super(setCat); }

    public abstract GO makeObj(SM source, SM target);

    protected GO coequalizer(GM g, GM h,
                             Coequalizer<SO, SM> nCE,
                             Coequalizer<SO, SM> eCE) {
        SM src = eCE.univ(nCE.mor().comp(g.codom().source()));
        SM tgt = eCE.univ(nCE.mor().comp(g.codom().target()));
        return makeObj(src, tgt);
    }
    ...
}
```

Fig. 13. Abstract class representing the category $\mathcal{G}raph$ of unlabeled graphs

```
private class GraphCoequalizer implements Coequalizer<GO, GM> {
    private final Coequalizer<SO, SM> nCE, eCE;
    private final GO obj;
    private final GM mor;

    private GraphCoequalizer(GM g, GM h) {
        nCE = setCat.coequalizer(g.nodeMor(), h.nodeMor());
        eCE = setCat.coequalizer(g.edgeMor(), h.edgeMor());
        obj = coequalizer(g, h, nCE, eCE);
        mor = makeMor(g.codom(), obj, nCE.mor(), eCE.mor());
    }
    public GO obj() { return obj; }
    public GM mor() { return mor; }
    public GM univ(GM m) {
        SM nm = nCE.univ(m.nodeMor());
        SM em = eCE.univ(m.edgeMor());
        return makeMor(obj, m.codom(), nm, em);
    }
}
```

Fig. 14. Inner class of GraphCat<SO,SM,GO,GM> computing coequalizers in graph-based categories

in GraphCat<SO,SM,GO,GM>. Fig. 14 shows class GraphCoequalizer; the others are very similar. The construction of the coequalizer morphism and the universal property directly follow from the description at the beginning of this section and shown in Fig. 11. However, the construction of the actual coequalizer objects differs for the different specific graph-based categories and, therefore, is

```
public class GraphImpl implements Graph<Set<?>, SetMor> {
    private final SetMor source, target;
    public GraphImpl(SetMor source, SetMor target) {
        assert source.dom().equals(target.dom());
        assert source.codom().equals(target.codom());
        this.source = source;   this.target = target;
    }
    public Set<?> nodes() { return source.codom(); }
    public Set<?> edges() { return source.dom(); }
    public SetMor source() { return source; }
    public SetMor target() { return target; }
}

public class GraphMorImpl
            implements GraphMor<SetMor>, Mor<GraphImpl, GraphMorImpl> {
    private final SetMor nodeMor, edgeMor;
    private final GraphImpl from, to;
    public GraphMorImpl(GraphImpl from, GraphImpl to,
                        SetMor nodeMor, SetMor edgeMor ) {
        assert nodeMor.dom().equals(from.nodes());
        assert nodeMor.codom().equals(to.nodes());
        assert edgeMor.dom().equals(from.edges());
        assert edgeMor.codom().equals(to.edges());
        this.from = from;              this.to = to;
        this.nodeMor = nodeMor;   this.edgeMor = edgeMor;
    }
    public SetMor edgeMor() { return edgeMor; }
    public SetMor nodeMor() { return nodeMor; }
    public GraphImpl dom() { return from; }
    public GraphImpl codom() { return to; }
    public GraphMorImpl comp(GraphMorImpl o) {
        SetMor nm = nodeMor().comp(o.nodeMor());
        SetMor em = edgeMor().comp(o.edgeMor());
        return new GraphMorImpl(o.dom(), codom(), nm, em);
    }
}

public class UGraphCatImpl
            extends UGraphCat<Set<?>, SetMor, GraphImpl, GraphMorImpl> {
    public static final UGraphCatImpl GRAPH = new UGraphCatImpl();

    private UGraphCatImpl() { super(SET); }
    public GraphImpl makeObj(SetMor source, SetMor target) {
        return new GraphImpl(source, target);
    }
    public GraphMorImpl makeMor(GraphImpl from, GraphImpl to,
                                SetMor nodeMor, SetMor edgeMor) {
        return new GraphMorImpl(from, to, nodeMor, edgeMor);
    }
}
```

Fig. 15. Concrete implementation classes for category $\mathcal{G}raph$

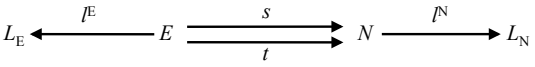
left to the abstract method coequalizer(...) that must be implemented by concrete subclasses of GraphCat<SO,SM,GO,GM>. Fig. 13 shows its implementation in class UGraphCat<SO,SM,GO,GM> for unlabeled graphs. It follows directly the description at the beginning of this section and shown in Fig. 11. Note that the coequalizer object is actually created by calling the object factory method of UGraphCat<SO,SM,GO,GM>.

Fig. 15 completes a prototype implementation of *Graph* by implementing classes for graph objects as well as graph morphisms, and a concrete subclass of UGraphCat<SO,SM,GO,GM>. Set<?> and SetMor are used as representations of *Set* objects and morphisms. Note that class GraphMorImpl implements both the graph morphism interface GraphMor<SetMor> and the categorical morphism interface Mor<GraphImpl,GraphMorImpl> and , thus, fulfills the constraints for type parameter GM of class GraphCat<SO,SM,GO,GM>. Class UGraphCatImpl provides the actual bindings for type parameters SO, SM, GO, as well as GM, and concrete factory methods. Class UGraphCatImpl is actually a singleton class. Its only instance is accessible through class variable GRAPH.

5 Labeled Graphs

We now extend graphs to labeled directed graphs where nodes are labeled with elements of a labeling alphabet L_{N} and edges with elements of a labeling alphabet L_{E}. A labeled directed graph over labeling alphabets $(L_{\mathrm{N}}, L_{\mathrm{E}})$, formally, is $G = (N, E, s, t, l^{\mathrm{N}}, l^{\mathrm{E}})$ where (N, E, s, t) is a directed graph and $l^{\mathrm{N}} : N \to L_{\mathrm{N}}$ as well as $l^{\mathrm{E}} : E \to L_{\mathrm{E}}$ are total functions that assign labels to nodes and edges, respectively. A labeled directed graph G, hence, is fully defined by its four functions $s, t, l^{\mathrm{N}}, l^{\mathrm{E}}$ as shown in the upper part of Fig. 16. This also gives rise to the representation of labeled graphs by interface LGraph<SO,SM> as an extension of the interface Graph<SO,SM> by just adding two labeling functions. A graph morphism $m : G_1 \to G_2$ for labeled directed graphs is simply a plain graph morphism consisting of two functions $m = (m_{\mathrm{N}}, m_{\mathrm{E}})$ mapping nodes and functions with the additional property that node and edge labels are preserved, i.e., $l_2^{\mathrm{N}} \cdot m_{\mathrm{N}} = l_1^{\mathrm{N}}$ and $l_2^{\mathrm{E}} \cdot m_{\mathrm{E}} = l_1^{\mathrm{E}}$ in Fig. 16. The same representation GraphMor<F> (Fig. 10) as for unlabeled graphs can be used.

Labeled graphs together with graph morphisms form a category *LGraph* that has all colimits. *LGraph* "inherits" all colimits from *Set* like category *Graph* of unlabeled graphs. The same constructions apply as for *Graph*. We describe the constructions of coequalizers. Fig. 17 shows labeled graphs $G_1 = (N_1, E_1, s_1, t_1, l_1^{\mathrm{N}}, l_1^{\mathrm{E}})$ and $G_2 = (N_2, E_2, s_2, t_2, l_2^{\mathrm{N}}, l_2^{\mathrm{E}})$ together with graph morphisms $g, h : G_1 \to G_2$ similar to Fig. 11. The unlabeled coequalizer graph $G_{CE} = (nCE, eCE, src, tgt)$ is constructed in the same way as described at the beginning of the previous section from the coequalizer sets nCE and eCE. The node labeling function for G_{CE} is constructed as follows: $l_2^{\mathrm{N}} \cdot g_{\mathrm{N}} = l_1^{\mathrm{N}} = l_2^{\mathrm{N}} \cdot h_{\mathrm{N}}$ holds since g and h are graph morphisms preserving labeling. The node labeling function for G_{CE} is the unique *Set* morphism $u_{\mathrm{N}} : nCE \to L_{\mathrm{N}}$ such that $u_{\mathrm{N}} \cdot mor_{\mathrm{N}} = l_2^{\mathrm{N}}$. Morphism u_{N} is computed using the universal property of the




```
public interface LGraph<S, F> extends Graph<S, F> {
    F nodeLabel();
    F edgeLabel();
}
```

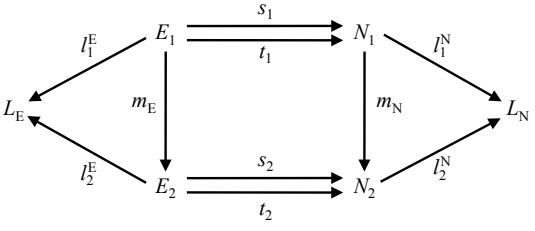


Fig. 16. Labeled directed graphs and their morphisms

Set coequalizer of g_N and h_N. The edge labeling of G_{CE} is constructed in a similar way.

Class GraphCat<SO,SM,GO,GM> (Fig. 12) is also an appropriate base-class of an implementation of category $\mathcal{LG}raph$. Type parameter GO, which represents graph objects, has to implement LGraph<SO,SM> now. Fig. 18 shows such an abstract subclass LGraphCat<SO,SM,GO,GM>. Each instance of the class is provided with the labeling alphabets of node and edge labels. Their sets are passed as constructor parameters and used for creating initial objects (not shown here). LGraphCat<SO,SM,GO,GM> provides an abstract factory method makeObj(...) for labeled graphs. Note that this method differs from the corresponding method in class UGraphCat<SO,SM,GO,GM> (Fig. 13) since it requires the labeling functions nodeLabel and edgeLabel as additional parameters. Finally, it provides

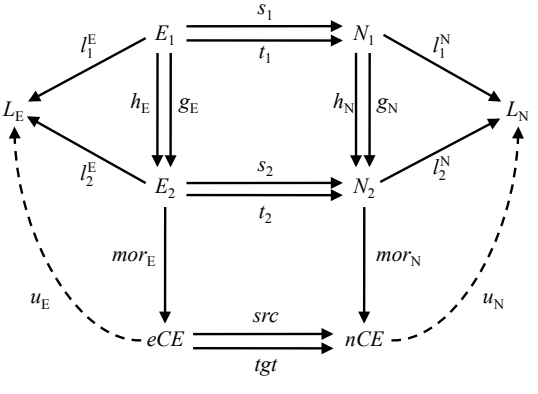


Fig. 17. Constructing coequalizers in $\mathcal{LG}raph$

```
public abstract class LGraphCat<SO, SM extends Mor<SO, SM>,
                                GO extends LGraph<SO, SM>,
                                GM extends GraphMor<SM> & Mor<GO, GM>>
                    extends GraphCat<SO, SM, GO, GM> {
    private final SO nodeLabels, edgeLabels;
    public LGraphCat(CatWithColimits<SO, SM> setCat,
                     SO nodeLabels, SO edgeLabels) {
        super(setCat);
        this.edgeLabels = edgeLabels;
        this.nodeLabels = nodeLabels;
    }

    public abstract GO makeObj(SM source, SM target,
                               SM nodeLabel, SM edgeLabel);

    protected GO coequalizer(GM g, GM h,
                             Coequalizer<SO, SM> nCE,
                             Coequalizer<SO, SM> eCE) {
        SM src = eCE.univ(nCE.mor().comp(g.codom().source()));
        SM tgt = eCE.univ(nCE.mor().comp(g.codom().target()));
        SM nLabel = nCE.univ(g.codom().nodeLabel());
        SM eLabel = eCE.univ(g.codom().edgeLabel());
        return makeObj(src, tgt, nLabel, eLabel);
    }
    ...
}
```

Fig. 18. Abstract class representing the category $\mathcal{LG}raph$ of labeled graphs

methods for computing colimit objects. Only the coequalizer computing method is shown in Fig. 18: Source and target morphism are computed in the same way as by the corresponding method in factory class UGraphCat<SO,SM,GO,GM>. The labeling functions are constructed in the same way as described above.

We omit the presentation of concrete classes for $\mathcal{LG}raph$ here. They can be easily built similar to or on top of the classes shown in Fig. 15 for $\mathcal{G}raph$.

6 Graph Transformations

The algebraic approach to graph transformation actually consists of two approaches, the so-called *double-pushout (DPO) approach* [7] and the *single-pushout (SPO) approach* [11].

A graph transformation rule p of the DPO approach (Fig. 19) consists of two total graph morphisms $l : K \to L$ and $r : K \to R$. If graph L can be found within the so-called *host* graph G, represented by the *match* morphism $m : L \to G$, then G can be *derived* to a graph H by applying p at m, written $G \overset{p,m}{\Longrightarrow} H$, if one can find a graph D and total graph morphisms $d : K \to D$, $l^* : D \to G$,

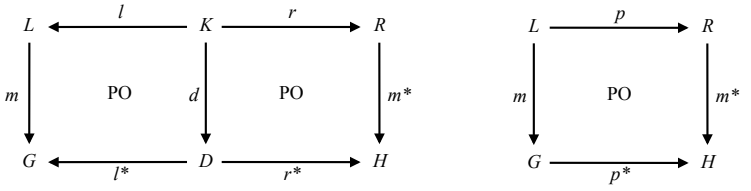


Fig. 19. Algebraic approach to graph transformation: DPO (left) and SPO (right)

$r^* : D \to H$, and $m^* : R \to H$ such that (m, l^*) is the pushout of (l, d) and (m^*, r^*) is the pushout of (r, d).

The SPO approach, on the other hand, is based on the concept of *partial* graph morphisms. We introduce *partial* functions first. A partial function $f : A \rightharpoonup B$ on finite sets is a total function $\mathrm{def}(f) \to B$ for a subset $\mathrm{def}(f) \subseteq A$; function f is not defined for any value $x \in A \setminus \mathrm{def}(f)$. A total function $f : A \to B$ is a partial function with $\mathrm{def}(f) = A$. Partial functions can be easily modeled by total functions by introducing the "undefined" value $\perp$ as a new value. Whenever a function f is actually undefined for a value x, we define $f(x) = \perp$. Moreover, we require $f(\perp) = \perp$ for each function f. In the following, we actually use total functions with the undefined value $\perp$ when considering partial functions.

A partial graph morphism $m : G \to H$ from a graph G to a graph H is a pair $m = (m_\mathrm{N}, m_\mathrm{E})$ of partial functions $m_\mathrm{N} : N_G \rightharpoonup N_H$ and $m_\mathrm{E} : E_G \rightharpoonup E_H$ that map nodes and edges, respectively, and preserve the structure of graphs, i.e., $s_H \cdot m_\mathrm{E}(e) = m_\mathrm{N} \cdot s_G(e)$ and $t_H \cdot m_\mathrm{E}(e) = m_\mathrm{N} \cdot t_G(e)$ for each edge $e \in \mathrm{def}(m_\mathrm{E})$.

The class of all graphs together with partial graph morphisms is again a category, called $\mathcal{G}raph^\mathcal{P}$, which has all colimits. We will discuss an implementation of this category in the following, based on the concepts presented in the previous sections. However, we have to define the SPO approach first:

A graph transformation rule p of the SPO approach (Fig. 19) consists of a single, but *partial* graph morphism $p : L \to R$. If graph L can be found within *host* graph G, represented by the total *match* morphism $m : L \to G$, then G can be *derived* to a graph H by applying p at m, written $G \overset{p,m}{\Longrightarrow} H$, by constructing the pushout (m^*, p^*) of (p, m). Graph H is then just the pushout graph.

Every realization of the DPO approach requires the construction of the so-called *pushout complement*: Instead of computing the pushout (m, l^*) of the morphisms (l, d), one has to compute (d, l^*) from (l, m) such that (m, l^*) is the pushout of (l, d). This is a non-trivial operation that even may have several non-isomorphic results [19]. The SPO approach, on the other hand, only requires constructing a pushout in $\mathcal{G}raph^\mathcal{P}$. The following shows that implementing $\mathcal{G}raph^\mathcal{P}$ is straightforward based on the concepts presented in the previous sections. We, therefore, present a realization of the SPO approach by realizing $\mathcal{G}raph^\mathcal{P}$ in the following. However, we do not present a solution for finding a match morphism of a production's left-hand side graph L to the host graph G; we rather assume that such a match morphism is provided, either manually, implicitly by the problem domain [19, Ch. 1], or by an existing graph matching algorithm (e.g., [1]).

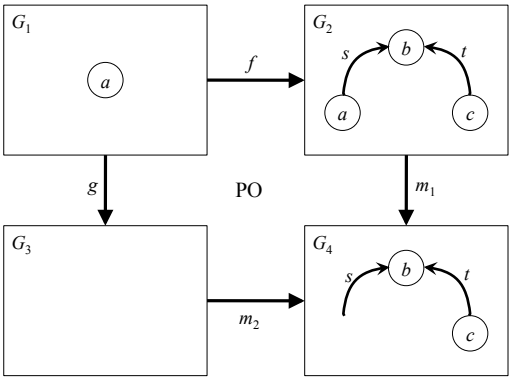


Fig. 20. Pushout construction by creating the pushout of node and edges sets separately in $\mathcal{S}et^{\mathcal{P}}$

Since the categories $\mathcal{G}raph$ and $\mathcal{LG}raph$ of unlabeled and labeled graphs with total graph morphisms are built upon the category $\mathcal{S}et$ of finite sets with total functions, we can try to build $\mathcal{G}raph^{\mathcal{P}}$ and $\mathcal{LG}raph^{\mathcal{P}}$ of unlabeled and labeled graphs with partial graph morphisms upon $\mathcal{S}et^{\mathcal{P}}$, the category of finite sets with partial functions. However, this construction is not completely correct: The constructions of coequalizers as described in the previous two sections and in Fig. 11 as well as Fig. 17 create also partial functions as source and target functions, i.e., the coequalizer "graph" may be no graph at all because it may contain *dangling edges* that do not have a source and/or target node. This is demonstrated by Fig. 20: (m_1, m_2) is the pushout of (f, g) where f maps node a in G_1 to node a in G_2, and g maps G_1 to the empty graph G_3, i.e., the node mapping of g is undefined for node a. We obtain structure G_4 when creating the pushouts of node and edge sets separately; the source function of G_4 is undefined for edge s, i.e., G_4 is not a graph. The correct solution in $\mathcal{G}raph^{\mathcal{P}}$ would be G_4 without dangling edge s. In the following, we call such "graphs" with partial source and target functions *pseudo graphs*. Of course, each graph is a pseudo graph, too. Morphisms between pseudo graphs are also very much like graph morphisms except that domains and codomains are pseudo graphs instead of graphs.

The solution to the problem of dangling edges, i.e., just deleting all dangling edges, is actually also the solution of building $\mathcal{G}raph^{\mathcal{P}}$ and all other graph-based categories on top of $\mathcal{S}et^{\mathcal{P}}$. It is sufficient to have initial object, binary coproducts and coequalizers in a category in order to have all finite colimits [18]; we used this for computing pushouts from binary coproducts and coequalizers (Fig. 7 and 8). Based on this construction, we have to remove dangling edges only when computing coequalizers because initial object and binary coproducts cannot produce dangling edges. Fig. 21 shows the resulting coequalizer construction: Pseudo graph G is constructed based on $\mathcal{S}et^{\mathcal{P}}$ by computing coequalizers of node and edge sets separately for morphisms g and h. Hence, G may contain dangling

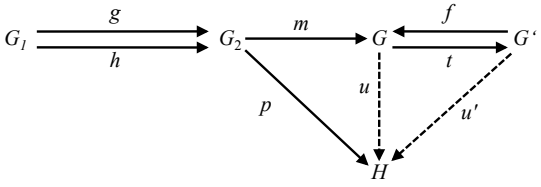


Fig. 21. Coequalizer construction in $\mathcal{G}raph^{\mathcal{P}}$ based on $\mathcal{S}et^{\mathcal{P}}$

edges. Graph G' is obtained from G by just deleting those dangling edges, i.e., it is a substructure of G, indicated by embedding morphism $f : G' \to G$. The partial morphism $t : G \to G'$ is its inverse. It is easy to show that G' together with morphism $t \cdot m$ is the coequalizer of morphisms g and h in $\mathcal{G}raph^{\mathcal{P}}$. The universal property construction of this coequalizer can also be derived: If H is a graph and $p : G_2 \to H$ a (partial) graph morphism such that $p \cdot g = p \cdot h$, there is a unique (partial) morphism $u : G \to H$ such that $u \cdot m = p$ because of the universal property of the coequalizer construction of G. Moreover, there is a graph morphism $u' : G' \to H$ such that $u' \cdot t \cdot m = p$, and $u' = u \cdot f$.

We use this construction to realize $\mathcal{G}raph^{\mathcal{P}}$ and any other graph-based category with all finite colimits. The first step is realizing $\mathcal{S}et^{\mathcal{P}}$, the category of finite sets with partial functions. We can actually use the implementation of $\mathcal{S}et$ discussed in Sec. 2 and 3 and introduce a special value UNDEF as a representation of $\perp$. Classes SetMor and in particular Partition for the implementation of coequalizers (Fig. 5) are modified such that $f(\perp) = \perp$ holds for each function f.

The next step is realizing the construction of coequalizers in $\mathcal{G}raph^{\mathcal{P}}$ and all other graph-based categories. Pseudo graphs are represented by the generic interface PseudoGraph<GO,GM> (Fig. 22). Type parameters GO and GM stand for the Java types of objects and morphisms in the corresponding category. Method isProperGraph() checks whether the represented pseudo graph is actually a graph, i.e., whether it does not contain any dangling edges. The other three methods represent the construction shown in Fig. 21 where G is the pseudo graph being represented by interface PseudoGraph<GO,GM>: properGraph() computes graph G' by just deleting all dangling edges from G; methods toProperGraph() and fromProperGraph() return morphisms $t : G \to G'$ and $f : G' \to G$, respectively.

The actual coequalizer construction as described above is represented by generic class ProperGraphCat<GO,GM>. Type parameters GO and GM stand for the Java types of objects and morphisms in the corresponding category of pseudo graphs. Note that GO is actually a bounded type parameter with PseudoGraph<GO,GM> being its upper bound, i.e., GO stands for any pseudo graph type. ProperGraphCat<GO,GM> realizes a category with colimits since it is a subclass of CocoCat<GO,GM>. Initial object and binary coproducts of this category are actually computed using the category being passed as constructor parameter. Coequalizers, however, are computed following the construction shown in Fig. 21 and using methods of PseudoGraph<GO,GM>.

```
public interface PseudoGraph<GO, GM> {
    boolean isProperGraph();
    GO properGraph();
    GM toProperGraph();
    GM fromProperGraph();
}

public class ProperGraphCat<GO extends PseudoGraph<GO, GM>,
                            GM extends Mor<GO, GM>>
        extends CocoCat<GO, GM> {
    private final CatWithColimits<GO, GM> cat;
    public ProperGraphCat(CatWithColimits<GO, GM> cat) { this.cat = cat; }

    public GM id(GO obj) { return cat.id(obj); }
    public InitialObj<GO, GM> initialObj() { return cat.initialObj(); }
    public Coproduct<GO, GM> coproduct(GO o1, GO o2) {
        return cat.coproduct(o1, o2);
    }
    public Coequalizer<GO, GM> coequalizer(GM f, GM g) {
        final Coequalizer<GO, GM> ce = cat.coequalizer(g, h);
        if (ce.obj().isProperGraph()) return ce;
        final GO obj = ce.obj().properGraph();
        final GM mor = ce.obj().toProperGraph().comp(ce.mor());
        return new Coequalizer<GO, GM>() {
            public GM mor() { return mor; }
            public GO obj() { return obj; }
            public GM univ(GM m) {
                return ce.univ(m).comp(ce.obj().fromProperGraph());
            }
        };
    }
}
```

Fig. 22. Creating graph-based categories on top of $\mathcal{S}et^{\mathcal{P}}$ using pseudo graphs

Using these concepts, we can represent category $\mathcal{G}raph^{\mathcal{P}}$ of unlabeled graphs with partial morphisms as follows: we assume that the realization of $\mathcal{S}et$ has already been extended to $\mathcal{S}et^{\mathcal{P}}$ as described above. The Java class of graph objects must be extended to implement interface PseudoGraph<GO,GM>, too, i.e., our example class GraphImpl (Fig. 15) must additionally implement PseudoGraph<GraphImpl,GraphMorImpl>. The actual implementation of the required methods is straightforward and omitted here. Category $\mathcal{G}raph^{\mathcal{P}}$ is then represented by an object obtained by creating a ProperGraphCat<GO,GM> instance:

```
new ProperGraphCat<GraphImpl, GraphMorImpl>(UGraphCatImpl.GRAPH)
```

where UGraphCatImpl.GRAPH is the instance of singleton class UGraphCatImpl (Fig. 15). Therefore, $\mathcal{G}raph^{\mathcal{P}}$ is actually implemented as described in Sec. 4.

Special actions, however, are taken if pseudo graphs with dangling edges are constructed. Those pseudo graphs are transformed into graphs by simply deleting dangling edges.

The same procedure is applicable to the category $\mathcal{LG}raph^{\mathcal{P}}$ of labeled graphs with partial morphisms: ProperGraphCat<GO,GM> simply must be instantiated with an instance of a class implementing LGraphCat<SO,SM,GO,GM>.

7 Conclusions

In this paper, we have presented how concepts of category theory can be represented and implemented in Java. Compared to the original implementation by Rydeheard and Burstall using the functional programming language ML, a much more modular implementation is provided using object-oriented implementation techniques and generic type parameters. Using these techniques, a complete implementation of the single-pushout approach to graph transformation has been presented. A graph matcher that finds occurrences of a graph within another graph is the only missing aspect for a complete graph transformation machine. However, one can easily make use of existing graph matching implementations to this end.

We have not touched the question of efficiency of the presented approach. Comparing it with other graph transformation implementations is worthwhile since other approaches like PROGRES and FUJABA have been realized with efficiency being one of their primary goals. Our primary goal was not efficiency, but an implementation technique that follows categorical definitions, constructions, and proofs as closely as possible. Yet, efficiency does not necessarily suffer. All of our implementations are generic in such a way that concrete implementations can be replaced by more efficient ones without much or even without any effort. The implementation of category $\mathcal{S}et$ is an example: We have presented a very simple and straightforward implementation, but we have actually implemented[7] (but not presented here) a much more efficient solution. For instance, the presented $\mathcal{S}et$ morphism composition in class SetMor (Fig. 2) has a runtime complexity that is linear in the size of the involved objects, even if assertion checking is switched off. We have realized an alternative implementation that maps set elements based on its position in an array. That allows for function composition and even building binary coproducts of sets as well as graphs in constant time. However, complexity of less trivial operations like coequalizers or pushouts and, hence, graph transformations are more expensive, even when not taking into account the most expensive operation – graph matching. One of the essential concepts of computational category theory is to not modify existing objects; morphisms being related to modified objects would become invalid otherwise. Performing graph transformations, hence, requires the construction of new graph objects. This is at least of linear complexity[8] in the size of the

[7] An extended version of the presented Java class library can be downloaded from `www.unibw.de/inf2/CatProg`

[8] Constructing the $\mathcal{S}et$ coequalizer based on the solution of the union-find problem has a complexity slightly worse then linear.

graphs involved, whereas an efficient graph transformation implementation like Fujaba modifies graphs in-place.

This paper has presented a realization of unlabeled as well as labeled graphs and graph transformations on them. Realization of further graph classes like structurally labeled graphs [16,20] or attributed graphs [6] is rather straightforward. Moreover, we will continue to represent other graph transformation approaches, e.g., Adaptive Star Grammars [5].

Acknowledgments. We thank the anonymous reviewers for their constructive reviews of this paper.

References

1. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 396–410. Springer, Heidelberg (2008)
2. Bracha, G.: The Java Tutorials – Lesson: Generics, http://java.sun.com/docs/books/tutorial/extra/generics (Last checked: March 12, 2010)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
4. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. In: Rozenberg [17], ch. 3, pp. 163–245
5. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Eetvelde, N.V.: Adaptive star grammars. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 77–91. Springer, Heidelberg (2006)
6. Ehrig, H., Heckel, R., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation part II: Single pushout approach and comparison to double pushout approach. In: Rozenberg [17], ch. 4, pp. 247–312
7. Ehrig, H., Pfender, M., Schneider, H.: Graph grammars: An algebraic approach. In: IEEE Conf. on Automata and Switching Theory, Iowa City, pp. 167–180 (1973)
8. Engels, G., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages, and Tools, vol. II. World Scientific, Singapore (1999)
9. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: Language and environment. In: Engels et al. [8], ch. 14, pp. 551–603
10. Herrlich, H., Strecker, G.E.: Category Theory - An Introduction. Allyn and Bacon, Boston (1973)
11. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science 109, 181–224 (1993)
12. MacLane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, vol. 5. Springer, New York (1971)
13. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer, Berlin (2008)
14. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (2002)

15. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996)
16. Parisi-Presicce, F., Ehrig, H., Montanari, U.: Graph rewriting with unification and composition. In: Ehrig, H., Nagl, M., Rozenberg, G., Rosenfeld, A. (eds.) Graph-Grammars and Their Application to Computer Science. LNCS, vol. 291, pp. 496–514. Springer, Heidelberg (1987)
17. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. I. World Scientific, Singapore (1997)
18. Rydeheard, D.E., Burstall, R.M.: Computational Category Theory. Prentice Hall, Englewood Cliffs (1988)
19. Schneider, H.J.: Graph transformations – An Introduction to the Categorical Approach. Preliminary version (October 2009), http://www2.informatik.uni-erlangen.de/staff/schneider/gtbook
20. Schneider, H.J.: Changing labels in the double-pushout approach can be treated categorically. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 134–149. Springer, Heidelberg (2005)
21. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In: Engels et al. [8], ch. 13, pp. 487–550
22. Zakhour, S., Hommel, S., Royal, J., Rabinovitch, I., Risser, T., Hoeber, M.: The Java Tutorial: A Short Course on the Basics, 4th edn. Prentice Hall PTR, Englewood Cliffs (2006)

On GS-Monoidal Theories for Graphs with Nesting

Roberto Bruni[1], Andrea Corradini[1], Fabio Gadducci[1], Alberto Lluch Lafuente[2], and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa, Italy
{bruni,andrea,gadducci,ugo}@di.unipi.it
[2] IMT Institute for Advanced Studies, Lucca, Italy
alberto.lluch@imtlucca.it

Abstract. We propose a sound and complete axiomatisation of a class of graphs with nesting and either locally or globally restricted nodes. Such graphs allow to represent explicitly and at the right level of abstraction some relevant topological and logical features of models and systems, including nesting, hierarchies, sharing of resources, and pointers or links. We also provide an encoding of the proposed algebra into terms of a gs-monoidal theory, and through these into a suitable class of "well-scoped" term graphs, showing that this encoding is sound and complete with respect to the axioms of the algebra.

1 Introduction

The use of graphs or diagrams of various kinds is pervasive in Computer Science, as they are very handy for describing in a two-dimensional space the logical or topological structure of systems, models, states, behaviours, computations, and several other entities of interest; the reader might be familiar, for example, with the graphical presentations of entity-relationship diagrams, of finite state automata, of static and behavioural UML diagrams (like class, message sequence and state diagrams), of computational formalisms like Petri nets, and so on.

The advantage of using graphs or diagrams, rather than a linear syntax based on terms or strings, lies in the fact that graphs can represent in a direct way relevant topological features of the systems/models they describe, including nesting, hierarchies, sharing of structures, and pointers or links, among others, making such features easily understandable also to non-specialists. In several cases graphs provide a representation of models or systems at the "right" level of abstraction: often a single graph corresponds to an equivalence class of terms, up to an axiomatic specification equating systems considered as topologically indistinguishable. Furthermore, as drawings are always understood "up to isomorphism", if the concrete identities of certain syntactical entities are irrelevant (for example, the name of the states of a finite state automata), it is sufficient to avoid depicting them in the drawing (a state is uniquely identified by the graphical components it is represented with).

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 59–86, 2010.

Another interesting case where a graphical syntax allows one to get rid of irrelevant information from the linear syntax is the representation of terms or formulæ of languages with binding operators (like first-order formulæ with quantifiers, λ-terms with abstractions, terms of process calculi with name restriction operators, among others). In all such cases, the classical linear syntax actually considers terms up to α-conversion (i.e., non-capturing renaming of bound variables or names): a graphical syntax can easily handle this, by representing a bound variable/name with an unlabelled node and references to it with edges.

Unfortunately, though, graphical representations are much more difficult to handle and to analyse than linear ones. In general, a graphical model needs to be encoded into a linear syntax, in order to exploit tools like theorem provers or model checkers to verify certain properties on it. Typically, a linear syntax can be defined by introducing an equational signature, whose operator symbols are interpreted as operations on graphs and where the axioms formalise suitable properties of these operators: then the terms of the initial algebra can be interpreted as graphs.

In this paper we are concerned with graphical notations that treat as first-class citizens the sharing of possibly bound names as sketched above, as well as the nesting of structures, a feature emerging as a recurrent pattern in the conceptual modelling of systems: on the informatics side, one may think of file systems, composite diagrams, networks, sessions, transactions, locations, structured state machines, or even XML documents, among others; or one may consider natural models of computations, like those arising in bioinformatics, equating nesting with the presence of membranes for molecules in chemical compounds.

As a main contribution, we introduce in Section 2 a visual modelling framework suited for representing systems exhibiting nesting of structures and sharing of atomic, named resources, as well as both local and global restriction. This framework consists not only of a class of hierarchical graphs, called *NR-graphs*, which allow to represent such systems in a direct, intuitive way, but also of a standard algebraic presentation, made of a signature and a set of axioms, defining the *algebra of graphs with nesting*, called **AGN**. The two components are related by a formal result stating that the proposed axiomatisation is sound and complete, i.e., that equivalence classes of terms of the algebra are in bijective correspondence with NR-graphs taken up to isomorphism: this relationship is represented by the top horizontal arrow of the next diagram.

$$\begin{array}{ccc} \mathbf{AGN}(S,B)/_{\equiv_{\mathcal{A}}} & \Longleftrightarrow & \text{NR-Graphs over } (S,B) \\ {\scriptstyle Sec.\,4}\big\downarrow & & \big\downarrow{\scriptstyle Sec.\,5} \\ \mathbf{GS}(\Sigma_B^\bullet) & \xleftarrow{[10]} & \text{Term Graphs over } \Sigma_B^\bullet \end{array}$$

This result is not proved directly, but it is obtained, indirectly, by relating the newly introduced framework of NR-graphs to the well-developed theory of *term graphs*. Roughly, term graphs, presented in Section 3, are directed acyclic graphs over a signature Σ, and they intuitively represent "terms with shared sub-terms" over Σ: therefore they are inherently simpler than our NR-graphs (they feature neither nesting nor restriction). An algebraic, equational characterisation of term

graphs was proposed in [10] exploiting the so-called *gs-monoidal theories*: it is analogous to the characterisation of terms built over a signature Σ as arrows of a cartesian category, the Lawvere theory of Σ, freely generated from the signature. The bijective correspondence between term graphs and equivalence classes of gs-monoidal terms is represented by the bottom arrow of the above diagram.

To relate the two formalisms, we first present in Section 4 an encoding of the terms of the algebra **AGN** into terms of the gs-monoidal theory (the left vertical arrow above): this translation is shown to be sound and complete, i.e., two terms are mapped to equivalent gs-monoidal terms if and only if they are equivalent. Intuitively, this is possible because the nesting of nodes and edges is encoded faithfully by exploiting a new sort of the signature of term graphs, introduced to represent *locations*; furthermore, global and local restriction are represented with suitable operators. Next in Section 5 we show that there is a bijective correspondence between NR-graphs and "well-scoped" term graphs (the right vertical arrow above): through the composition of this bijection with the encoding of **AGN** terms as gs-monoidal terms we obtain the bijection between the terms of the algebra and NR-graphs.

The bridge between the theories of graphs with nesting and of term graphs established by the proposed encoding can be used to exploit in the newly introduced framework the rich theory and pragmatics developed for term graphs and term graph rewriting along the years: we sketch some possible developments in Section 6. We conclude by reviewing some relevant related works in Section 7 and by proposing some topics of future research in Section 8.

2 An Algebra for Graphs with Nesting and Restrictions

Many notions of hierarchically structured graphs were proposed in the literature (see Section 7), mostly defined set-theoretically as (hyper-)graphs whose nodes and edges can be related to other graphs (e.g., by suitable containment morphisms or adjacency relations). Rarely such graphs come equipped with a handy syntax for representing and manipulating them algebraically, a topic on which we recently made some progress, building on previously developed notations like CHARM [11] and other syntactic formalisms for representing plain graphs with interfaces [14]. We have been also influenced by the notation used in nominal calculi.

In the present section we introduce the *graphs with nesting and restriction*, briefly *NR-graphs*, as well as an equational axiomatisation for them: the main result states that equivalence classes of terms of the algebra modulo the axioms are in bijective correspondence with isomorphism classes of graphs. The hierarchical graphs we present are based on the following rationale: 1) for flexibility and visual expressiveness we prefer to deal with hyper-graphs instead of ordinary two-ended arcs; 2) nesting is seen according to a classical boxes-within-boxes scheme and applied to edges, while nodes are seen mainly as attaching points without further containment; 3) nodes can be localised within a single edge, in

which case they are made private and not visible "outside"; 4) nodes can also be globally available to allow connections across the nesting hierarchy.

2.1 Graphs with Nesting and Restriction

Let us introduce some formal notation. Given a set M we denote by M^* the free monoid over M, i.e., the set of finite lists of elements drawn from M, often denoted by an over-lined letter. The unit of M^* is denoted by ϵ. Depending on the context, we can find it more convenient to denote concatenation just by juxtaposition (like in $\overline{e} = e_1 e_2 ... e_n$) or with commas (like in $\overline{e} = e_1, e_2, ..., e_n$). Given a list $\overline{e}$ we denote by $\overline{e}|_i$ its i-th element and by $|\overline{e}|$ the underlying set of list elements. For an ordinal n, we write $\underline{n}$ for the set $\{1, \ldots, n\}$. We overload $\#\_$ to denote both the length of a list and the cardinality of a set. We use the symbol $\uplus$ for the disjoint union of sets. We write $s \in \overline{e}$ as a shorthand for $s \in |\overline{e}|$. If $f\colon A \to B$ is a function, we denote by f^* its obvious monoidal extension $f^*\colon A^* \to B^*$ and by f itself the power-set extension $f\colon 2^A \to 2^B$.

All along this section, let S be a set of sorts, B be a ranked set of *box labels* with $rnk(b) \in S^*$ for all $b \in B$, and $\mathcal{X}$ be a countable set of names. A *node* is a pair $x : s \in \mathcal{X} \times S$, and the operator $\tau : (\mathcal{X} \times S)^* \to S^*$ applied to a list of nodes returns the list of their sorts. Note that nodes with the same name but with different sorts are kept distinct, e.g., $x : s_1 \neq x : s_2$ if $s_1 \neq s_2$.

We introduce now the formal definition of our hierarchical graphs. We define the set **NR-Graph** of *graphs with nesting and restriction* and, for a set X of nodes, the set **NR-Graph**$[X]$ of such graphs *with external nodes* X.

Definition 1 (NR-graphs). *An* NR-graph $G \in$ **NR-Graph** *is a tuple* $G = \langle FN, GR, H \rangle$*, where* FN *is a set of* free nodes*,* GR *is a set of* globally restricted nodes *with* $FN \cap GR = \emptyset$*, and* $H \in$ **NR-Graph**$[FN \cup GR]$*. The set of* global nodes *of* G *is given by* $FN \cup GR$*.*

An NR-graph H with external nodes X*,* $H \in$ **NR-Graph**$[X]$*, is a tuple* $H = \langle LR, E, l, c, \rho \rangle$*, where* LR *is a set of* locally restricted nodes *(satisfying* $LR \cap X = \emptyset$*),* E *is a set of (hyper-)edges,* $l\colon E \to B$ *labels each edge with an element of* B*,* $c\colon E \to (X \cup LR)^*$ *is the* connection function *(satisfying* $\tau(c(e)) = rnk(l(e))$ *for all* $e \in E$*), and* $\rho\colon E \to$ **NR-Graph**$[X \cup LR]$ *maps each edge to a graph nested within it.*

The *depth* of an NR-graph with external nodes $H = \langle LR, E, l, c, \rho \rangle$ is 0 if E is empty, and $depth(H) = 1 + max_{e \in E}\{depth(\rho(e))\}$ otherwise. The depth of an NR-graph $G = \langle FN, GR, H \rangle$ is the depth of H. We will consider only graphs of finite depth and with finite sets of edges and nodes.

Figure 1 shows a sample NR-graph of depth 3 which represents a network system comprising different subnets (net-labelled edges) and workstations (st) connected through links according to different patterns: each subnet has a single access point, while each workstation is attached to two connection hubs (s-labelled nodes). The top level of the graph is defined as $G = \langle \{x : s\}, \{y : s\}, H_0 \rangle$, with $H_0 = \langle \emptyset, \{e\}, \{e \mapsto \mathsf{net}\}, \{e \mapsto y\}, \{e \mapsto H_1\} \rangle$. G and H_0 define the global

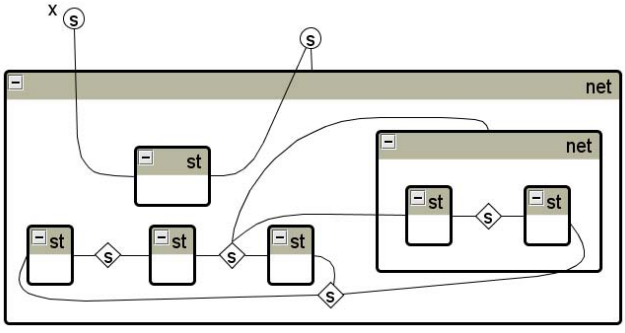


Fig. 1. A sample NR-graph, called G

nodes and the outer net-labelled edge, while H_1, which is going to be defined immediately below, represents the graph internal to edge e.

Note that the global nodes are depicted, with a round shape, at the top of the graph, as conceptually they do not pertain to any particular location. Also the name is depicted for free nodes (only $x : s$, in this case), but not for globally restricted nodes (like y). All nodes are sorted: the sort is indicated by the inside label, and in this example all nodes have the same sort s, therefore sometimes we will omit it. Here is part of the definition of graph H_1:

$$\langle \{z_1, z_2, z_3\}, \{f_1, f_2, f_3, f_4, f_5\}, \{f_1 \mapsto \mathsf{st}, \ldots\}, \{f_1 \mapsto x \cdot y, \ldots\}, \{f_1 \mapsto \emptyset_{NR}, \ldots\} \rangle$$

For the sake of brevity, we omit the rest of the definition of the NR-graph, but it should be clear from the drawing. H_1 has three locally restricted nodes, that are depicted as diamonds: they are private to the immediately enclosing edge e and cannot be referenced from the outside, but they can be shared by the subgraphs nested within the edge. H_1 also contains five edges: f_1 is the one labelled by st, connected to the two global nodes, and not containing anything, which is represented by $\rho(f_1) = \emptyset_{NR}$ ($\emptyset_{NR}$ is the empty NR-graph). Notice that edges are represented as ranked boxes, possibly nested, with a label in the upper-right corner. The rank information consists of the list of sorted tentacles attached to the box (where the sort of the tentacle is the sort of the node it is attached to). The ordering of the tentacles is left implicit by counting in clockwise order, starting from the leftmost tentacle. Nesting of edges and nodes within other edges is given by spatial containment.

Actually, Fig. 1 does not show precisely the above defined NR-graph G, but rather its isomorphism class, according to the definition that follows.

Definition 2 (NR-Graph isomorphism). *Let $G = \langle FN, GR, H \rangle$ and $G' = \langle FN', GR', H' \rangle$ be two NR-graphs. We say that G and G' are* isomorphic*, written $G \cong G'$, if $FN = FN'$, there is an isomorphism $\phi\colon FN \cup GR \to FN' \cup GR'$ such that $\phi(x) = x$ for all $x \in FN$, and H is ϕ-isomorphic to H', written $H \cong_\phi H'$.*

Let X and X' be two sets of nodes, and $\phi\colon X \to X'$ be an isomorphism. Furthermore, let $H = \langle LR, E, l, c, \rho \rangle$ be in **NR-Graph**$[X]$ *and $H' = \langle LR', E', l', c', \rho' \rangle$*

be in **NR-Graph**$[X']$. *Then* $H \cong_\phi H'$ *if there exist isomorphisms* $\phi_L \colon LR \to LR'$ *and* $\phi_E \colon E \to E'$ *such that, calling* $\hat{\phi} \colon X \cup LR \to X' \cup LR'$ *the isomorphism induced by* ϕ *and* ϕ_L*, for all* $e \in E$ *it holds*

- $l'(\phi_E(e)) = l(e)$,
- $c'(\phi_E(e)) = \hat{\phi}^*(c(e))$, *and*
- $\rho'(\phi_E(e)) \cong_{\hat{\phi}} \rho(e)$.

Thus isomorphisms preserve the identity of free nodes, but not the one of restricted nodes and edges: this explains why only the identities of free nodes are depicted in Fig. 1.

2.2 The Algebra for Graphs with Nesting

Even if the graphical representation of a system like the one of Fig. 1 is pretty intuitive and easy to understand for human beings, it might not be usable, e.g., as the input of a verification tool needed to analyse it. On the other hand, the set-theoretical presentation according to Definition 1 does provide a linear syntax for such graphs, but it is quite involved, as it emerges from the (partial) definition given by the graphs G, H and H_1 above. The main motivation of the graph algebra we are going to introduce is to provide a much more compact and intuitively understandable linear syntax for NR-graphs.

Definition 3 (algebra for graphs with nesting and restrictions, AGN). *The terms of the* algebra for graphs with nesting *(or* nested graphs*) are generated according to the following grammar*

$$\mathbb{G} \ ::= \ \mathbf{0} \ \mid \ x : s \ \mid \ b[\mathbb{G}](\overline{y}) \ \mid \ \mathbb{G} \mid \mathbb{G} \ \mid \ (\nu\, x : s)\mathbb{G} \ \mid \ (\mu\, x : s)\mathbb{G}$$

where $x : s$ *is a node,* $b \in B$*, and* $\overline{y} \in (\mathcal{X} \times S)^*$ *is a list of nodes such that* $rnk(b) = \tau(\overline{y})$.

Roughly, $\mathbf{0}$ denotes the empty graph; $x : s$ is a discrete graph with a single node named x of sort s; $b[\mathbb{G}](\overline{y})$ is a hyper-edge labelled b, whose tentacles are attached to nodes $\overline{y}$ and enclosing the graph $\mathbb{G}$; $\mathbb{G} \mid \mathbb{H}$ is the disjoint union of graphs $\mathbb{G}$ and $\mathbb{H}$ up to common (free) nodes; finally, $(\nu\, x : s)\mathbb{G}$ and $(\mu\, x : s)\mathbb{G}$ denote the graph $\mathbb{G}$ after making node $x : s$ not visible from the outside (borrowing nominal calculus jargon, we say that the node $x : s$ is *restricted*). An **AGN** term where no edge $b[\mathbb{G}](\overline{y})$ appears is called *discrete* and usually denoted by $\mathbb{D}$. Recall that each label $b \in B$ has a fixed rank $rnk(b) \in S^*$: we only allow *well-sorted* graphs, where for any sub-term $b[\mathbb{G}](\overline{y})$ we have that the (lists of) sorts of b and $\overline{y}$ coincide (as required by the constraint $rnk(b) = \tau(\overline{y})$ in Definition 3).

Notably, we distinguish two kinds of restrictions: $(\mu\, x : s)\mathbb{G}$ is called *localised restriction*, meaning that the node $x : s$ resides together with the topmost edges of $\mathbb{G}$, while $(\nu\, x : s)\mathbb{G}$ is called *global restriction*, meaning that the location of $x : s$ is immaterial. The key difference is that when a graph is enclosed within an edge, its globally restricted nodes can traverse up the hierarchy (see axiom A8

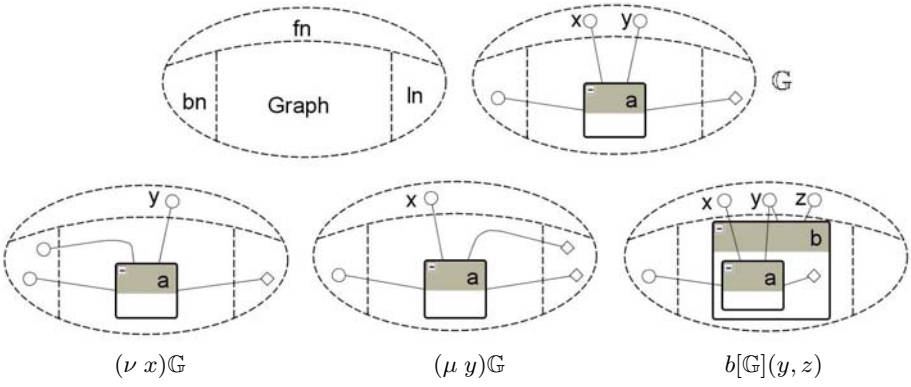


Fig. 2. Restrictions and nesting illustrated schematically

in Definition 5), while this is not the case for locally restricted nodes. When it is not necessary to distinguish which kind of restriction is considered, we write $(\omega\ x : s)$ using the wildcard $\omega \in \{\nu, \mu\}$. An **AGN** term where global restriction does not occur is called ν-*free*. Restrictions $(\nu\ x{:}s)\mathbb{G}$ and $(\mu\ x{:}s)\mathbb{G}$ act as binders for $x : s$ in $\mathbb{G}$, leading to the ordinary notion of *free nodes*.

Definition 4 (free nodes). *The set of* free nodes *of an* **AGN** *term* $\mathbb{G}$*, denoted* $fn(\mathbb{G})$*, is defined inductively as follows:*

$$fn(\mathbf{0}) \triangleq \emptyset \qquad fn(x : s) \triangleq \{x : s\} \qquad fn(b[\mathbb{G}](\overline{y})) \triangleq fn(\mathbb{G}) \cup |\overline{y}|$$

$$fn(\mathbb{G} \mid \mathbb{H}) \triangleq fn(\mathbb{G}) \cup fn(\mathbb{H}) \qquad fn((\omega\ x : s)\mathbb{G}) \triangleq fn(\mathbb{G}) \setminus \{x : s\}$$

As useful shorthands, we shall write $b(\overline{y})$ instead of $b[\mathbf{0}](\overline{y})$ and $b[\mathbb{G}]$ instead of $b[\mathbb{G}]()$: intuitively, the former denotes a plain edge, while the latter denotes a "floating" box (not anchored to any node). Moreover, we write $\prod_{i=1}^{n} \mathbb{G}_i$ as a shorthand for $\mathbb{G}_1 \mid (\mathbb{G}_1 \mid (\ldots \mid \mathbb{G}_n)\ldots))$ and we let $(\omega\ \overline{y})\mathbb{G}$ stand for the term $(\omega\ \overline{y}|_1)(\omega\ \overline{y}|_2)...(\omega\ \overline{y}|_m)\mathbb{G}$, where $m = \#\overline{y}$.

Figures 2 and 3 show the general idea for interpreting the operators of our algebra. We depict a graph as a large oval (see Fig. 2, top-left), with separated sectors for free nodes (top sector), globally (ν-) restricted nodes (left sector), top-level locally (μ-) restricted nodes (right sector) and all other nodes and edges (central sector). This is exemplified by a schematic graph drawn in Fig. 2, top-right, with a single edge attached to a few representative nodes (at least one of each kind, omitting their sorts): let us call it $\mathbb{G}$. The second line of Fig. 2 shows the graphs $(\nu\ x)\mathbb{G}$ (node x is moved from the sector of free nodes to that of globally restricted nodes), $(\mu\ y)\mathbb{G}$ (node y is moved from the sector of free nodes to that of locally restricted nodes) and $b[\mathbb{G}](y, z)$ (free nodes are shared between the enclosing edge and graph $\mathbb{G}$, globally bound nodes are preserved, localised nodes are enclosed in the top edge, leaving the right sector empty). Figure 3 shows the parallel composition of two generic graphs, obtained by taking the union of their free nodes and the disjoint union of all the other elements.

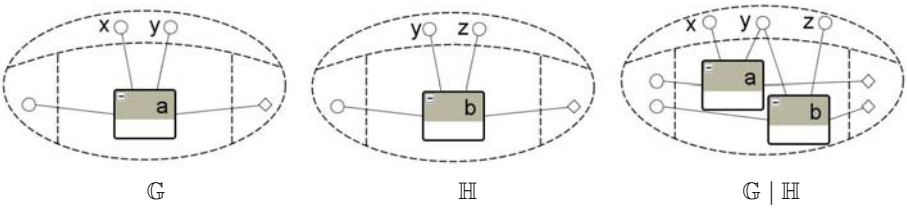


Fig. 3. Parallel composition illustrated schematically

Example 1. Figure 4 shows some graphs corresponding to simple terms of our algebra. Starting from top-left and in left-to-right reading direction we find the discrete graph $x : s$, the ordinary plain graphs $\mathsf{st}(x : s, y : s)$ and $\mathbb{G}_1 \triangleq \mathsf{st}(x, y) \mid \mathsf{st}(y, z)$, the graphs with restricted nodes $(\nu\ y)\mathbb{G}_1$ and $(\mu\ y)\mathbb{G}_1$, and the graphs with nesting $\mathsf{net}[(\nu\ y)\mathbb{G}_1](z)$ and $\mathsf{net}[(\mu\ y)\mathbb{G}_1](z)$.

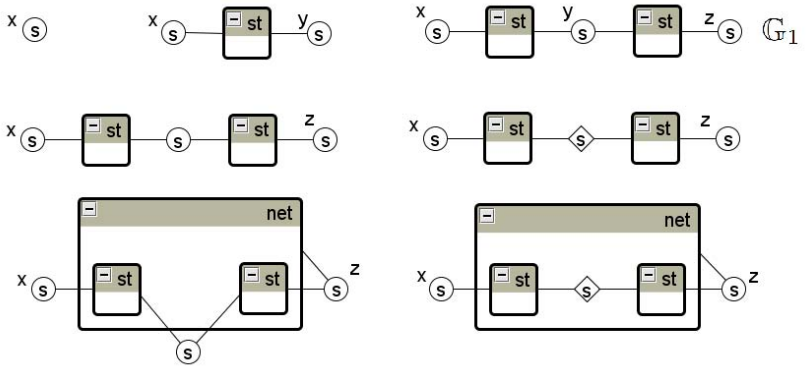


Fig. 4. Simple examples: $x{:}s$ (top-left), $\mathsf{st}(x{:}s, y{:}s)$ (top-centre), $\mathbb{G}_1 \triangleq \mathsf{st}(x, y) \mid \mathsf{st}(y, z)$ (top-right), $(\nu\ y)\mathbb{G}_1$ (mid-left), $(\mu\ y)\mathbb{G}_1$ (mid-right), $\mathsf{net}[(\nu\ y)\mathbb{G}_1](z)$ (bottom-left), $\mathsf{net}[(\mu\ y)\mathbb{G}_1](z)$ (bottom-right)

The terms of our algebra are too concrete, in the sense that different terms may intuitively correspond to the same nested graph (for example, the order in which we list the edges is obviously immaterial in the graph). Next we provide an axiomatisation equating those terms that define essentially the same graph.

The axiomatisation includes the structural graph axioms of [11] such as associativity and commutativity for $\mid$ with identity $\mathbf{0}$ (axioms A1–A3) and node restriction binding (A4–A6). It additionally includes axioms to α-rename bound nodes (A7), an axiom for the extrusion of globally bound, nested nodes (A8) that marks the distinction between global restriction ν and local restriction μ, an axiom for making immaterial the addition of a node to a graph where that same node is already free (A9) and an axiom ensuring that global nodes are not localised in lower layers (A10).

Definition 5 (structural congruence $\equiv_{\mathcal{A}}$). *The structural congruence $\equiv_{\mathcal{A}}$ over nested graphs is the least congruence satisfying*

$$\begin{aligned}
\mathbb{G} \mid \mathbb{H} &\equiv \mathbb{H} \mid \mathbb{G} && && \text{(A1)}\\
\mathbb{G} \mid (\mathbb{H} \mid \mathbb{I}) &\equiv (\mathbb{G} \mid \mathbb{H}) \mid \mathbb{I} && && \text{(A2)}\\
\mathbb{G} \mid \mathbf{0} &\equiv \mathbb{G} && && \text{(A3)}\\
(\omega_1\ x : s)(\omega_2\ y : t)\mathbb{G} &\equiv (\omega_2\ y : t)(\omega_1\ x : s)\mathbb{G} && \textit{if } x : s \neq y : t && \text{(A4)}\\
(\omega\ x : s)\mathbf{0} &\equiv (\omega\ x : s)x : s && && \text{(A5)}\\
\mathbb{G} \mid (\omega\ x : s)\mathbb{H} &\equiv (\omega\ x : s)(\mathbb{G} \mid \mathbb{H}) && \textit{if } x : s \notin \mathit{fn}(\mathbb{G}) && \text{(A6)}\\
(\omega\ x : s)\mathbb{G} &\equiv (\omega\ y : s)(\mathbb{G}\{^{y:s}/_{x:s}\}) && \textit{if } y : s \notin \mathit{fn}(\mathbb{G}) && \text{(A7)}\\
b[(\nu\ x : s)\mathbb{G}](\overline{y}) &\equiv (\nu\ x : s)b[\mathbb{G}](\overline{y}) && \textit{if } x : s \notin |\overline{y}| && \text{(A8)}\\
x : s \mid \mathbb{G} &\equiv \mathbb{G} && \textit{if } x : s \in \mathit{fn}(\mathbb{G}) && \text{(A9)}\\
b[x : s \mid \mathbb{G}](\overline{y}) &\equiv x : s \mid b[\mathbb{G}](\overline{y}) && && \text{(A10)}
\end{aligned}$$

where $\{^{y:s}/_{x:s}\}$ denotes the capture-avoiding substitution of $x : s$ by $y : s$ and $\omega, \omega_1, \omega_2$ range over $\{\nu, \mu\}$.

It is immediate to observe that structural congruence respects free nodes, i.e., $\mathbb{G} \equiv_{\mathcal{A}} \mathbb{H}$ implies $\mathit{fn}(\mathbb{G}) = \mathit{fn}(\mathbb{H})$ for any $\mathbb{G}, \mathbb{H}$.

Next statement establishes the soundness and the completeness of the proposed axiomatisation: the proof is based on the results presented in later sections.

Theorem 1. *The equivalence classes of terms of algebra* **AGN** *modulo $\equiv_{\mathcal{A}}$ are in bijective correspondence with the isomorphism classes of NR-graphs.*

Proof. The statement will follow from Propositions 2, 3 and 4.

The translation of **AGN** terms into NR-graphs is sketched above in Figs. 2 and 3. Vice versa, the intuitive way to express a nested graph as an **AGN** term is to start writing the discrete term corresponding to the free nodes of the graph, then to add arbitrary distinct names for the top-level unnamed nodes, with the corresponding ν- and μ-restrictions, and finally to list all the top-level edges, properly attached to the available nodes, and with this procedure applied inductively to the contents of each edge.

To conclude this section, let us show how the proposed algebra meets the goal of providing a concise and intuitive linear syntax for NR-graphs.

Example 2. The graph G in Fig. 1 corresponds to the following term $\mathbb{G}_G$ (where we omit the node sorts, all equal to s)

$$\begin{aligned}(\nu\ y)\mathsf{net}[\ \mathsf{st}(x, y)\ \mid\ (\mu\ z_1, z_2, z_3)(\ \mathsf{st}(z_3, z_1)\ \mid\ \mathsf{st}(z_1, z_2)\ \mid\ \mathsf{st}(z_2, z_3)\ \mid\\ \mathsf{net}[\ (\mu\ z_4)(\ \mathsf{st}(z_2, z_4)\ \mid\ \mathsf{st}(z_4, z_3)\ )\ ](z_2)\ )\ ](y)\end{aligned}$$

2.3 A Normalised Form for Terms of AGN

The axioms we just presented allow us to standardise the term-like representation of nested graphs, by transforming them into an equivalent normalised form. This form is not unique in general, but the equivalence among terms in this form can be characterised precisely by the existence of a structural bijection among them, as explained below.

Definition 6 (normalised form). *A term* $\mathbb{G}$ *is in* normalised form *if either it is* $\mathbf{0}$*, or it has the shape*

$$(\nu\ \overline{y})(\mu\ \overline{z})(\ \prod_{i=0}^{n} x_i : s_i\ \mid\ \prod_{j=0}^{m} b_j[\mathbb{G}_j](\overline{y}_j)\ )$$

where $n + m > 0$*, all nodes in* $\overline{y}$ *and* $\overline{z}$ *are pairwise distinct, all terms* $\mathbb{G}_j$ *for* $j \in \underline{m}$ *are* ν*-free and normalised themselves and, letting* $X \triangleq \bigcup_{i=1}^{n}\{x_i : s_i\}$*, we have* $\#X = n$*,* $fn(\mathbb{G}) \cup |\overline{y}| \cup |\overline{z}| = X$*, and* $fn(\mathbb{G}_j) = X$ *for all* $j \in \underline{m}$*.*

Proposition 1 (normalised form). *For any* **AGN** *term* $\mathbb{G}$ *it is possible to find a* $\equiv_{\mathcal{A}}$*-equivalent term* $\mathbb{H}$ *in normalised form.*

Proof (sketch). Roughly, the normalisation proceeds by first α-renaming all the restricted nodes so to make them pairwise distinct and also distinct from all the free nodes (axiom A7). Then all the restrictions are moved towards the top by applying axioms A4–A6 and A8. Note that while any ν-restriction can reach the top of the term (by A8), μ-restrictions cannot escape from their enclosing edge. Then, axioms A9–A10 are used to "saturate" each subgraph with all nodes available. For this task, we point out that $(\omega\ x : s)\mathbb{G} \equiv_{\mathcal{A}} (\omega\ x : s)(x : s \mid \mathbb{G})$: in fact, this property is trivial if $x : s \in fn(\mathbb{G})$ (by axiom A9), while otherwise it follows from

$$\begin{array}{rll}
(\omega\ x : s)\mathbb{G} & \equiv_{\mathcal{A}} (\omega\ x : s)(\mathbb{G} \mid \mathbf{0}) & \text{(by axiom A3)} \\
& \equiv_{\mathcal{A}} \mathbb{G} \mid (\omega\ x : s)\mathbf{0} & \text{(by axiom A6)} \\
& \equiv_{\mathcal{A}} \mathbb{G} \mid (\omega\ x : s)x : s & \text{(by axiom A5)} \\
& \equiv_{\mathcal{A}} (\omega\ x : s)(\mathbb{G} \mid x : s) & \text{(by axiom A6)} \\
& \equiv_{\mathcal{A}} (\omega\ x : s)(x : s \mid \mathbb{G}) & \text{(by axiom A1)}
\end{array}$$

Finally, we exploit axioms A1–A3 to properly rearrange the order of subgraphs composed in parallel, according to the shape of the normalised form. □

Example 3. The graph in Fig. 1 can be written in normalised form as the **AGN** term $(\nu\ y)(\ x \mid y \mid \mathsf{net}[(\mu\ z_1, z_2, z_3)(\mathbb{D}'_1 \mid \mathbb{G}'_1)](y)\ )$, where

$$\begin{array}{l}
\mathbb{D}'_1 \triangleq x \mid y \mid z_1 \mid z_2 \mid z_3 \\
\mathbb{G}'_1 \triangleq \mathsf{st}[\mathbb{D}'_1](x, y) \mid \mathsf{st}[\mathbb{D}'_1](z_3, z_1) \mid \mathsf{st}[\mathbb{D}'_1](z_1, z_2) \mid \mathsf{st}[\mathbb{D}'_1](z_2, z_3) \mid \mathsf{net}[\mathbb{G}_2](z_2) \\
\mathbb{G}_2 \triangleq (\mu\ z_4)(\ \mathbb{D}'_2 \mid \mathsf{st}[\mathbb{D}'_2](z_2, z_4) \mid \mathsf{st}[\mathbb{D}'_2](z_4, z_3)\ ) \\
\mathbb{D}'_2 \triangleq x \mid y \mid z_1 \mid z_2 \mid z_3 \mid z_4
\end{array}$$

Clearly, the normalised form of a term is not unique, not only because of α-conversion (axiom A7) but also because of the AC axioms for $\mid$ (A1 and A2) and of axiom A4, which allows to switch restrictions of the same type in an arbitrary way; nevertheless, it can be shown that these are the only sources of non-uniqueness. In fact, it is tedious but not difficult to prove that two terms $\mathbb{G}$ and $\mathbb{H}$ in normalised form are equivalent *if and only if* they have the same free nodes, and a suitable partial bijection ϕ can be established between the

sets of nodes of their corresponding syntax trees. Quite informally, ϕ must relate nodes with corresponding B-labelled boxes and μ- and ν-restrictions, preserving the nesting w.r.t. B-labelled boxes: essentially it records the permutations that can be applied to μ- and ν-restrictions of $\mathbb{G}$ (using A4) and to B-labelled boxes (using A1 and A2) in order to transform $\mathbb{G}$ into $\mathbb{H}$ (up to α-conversion).

The characterisation of the $\equiv_{\mathcal{A}}$-equivalence by the existence of a partial bijection is exploited in Section 4 when arguing about the completeness of the encoding of nested graphs into term graphs.

3 Term Graphs and GS-Monoidal Theories

This section introduces term graphs over a signature Σ as models of the gs-monoidal theory over Σ, by slightly generalising the main result of [10]: in fact, we shall consider many-sorted signatures instead of standard one-sorted ones.

Term graphs are defined as isomorphism classes of (ranked) directed acyclic graphs. Our main concern here is to stress the underlying algebraic structure, hence the presentation of term graphs slightly departs from the standard definition. With respect to the way term graphs are defined in the seminal paper [2], the main differences consist in the restriction to the acyclic case and the handling of empty nodes. A discussion about the relationship between the categorical and the traditional definition of term graphs can be found in [10].

Definition 7 (signature). *Given a set S of sorts, a* signature *Σ over S is a family $\{\Sigma_{u,s}\}_{u\in S^*,s\in S}$ of sets of operator symbols. For an operator $f \in \Sigma_{u,s}$, we call u its* arity *and s its* coarity*; sometimes we shall denote it as $f\colon u \to s$.*

Definition 8 (labelled graphs). *Let Σ be a signature over a set of sorts S. A* labelled (hyper-)graph *d* (over Σ) *is a tuple $d = \langle N, E, l_N, l_E, src, trg\rangle$, where N is a finite set of* nodes*, E is a finite set of* edges*, $src\colon E \to N^*$, $trg\colon E \to N$ are the* source *and* target connection functions*, and $l_N\colon N \to S$, $l_E\colon E \to \Sigma$ are the* labelling functions*, colouring nodes with sorts and edges with operator symbols. Furthermore, the following conditions must be satisfied*

1. *the connection functions are required to be consistent with the labelling, i.e., for all $e \in E$, $l_E(e) \in \Sigma_{u,s} \Leftrightarrow l_N^*(src(e)) = u \wedge l_N(trg(e)) = s$;*
2. *each node is the target of at most one edge, i.e., for all $e_1, e_2 \in E$, $trg(e_1) = trg(e_2) \Rightarrow e_1 = e_2$.*

A node n is *empty* if there is no edge $e \in E$ such that $n = trg(e)$; we shall denote by $N_\emptyset$ and N_Σ the sets of empty and non-empty nodes, respectively (thus $N = N_\Sigma \uplus N_\emptyset$). A labelled graph d is *discrete* if $E = \emptyset$. A *path* in d is a sequence $\langle n_0, e_0, n_1, \ldots, e_{m-1}, n_m\rangle$, where $m \geq 0$, $n_0, \ldots, n_m \in N$, $e_0, \ldots, e_{m-1} \in E$, and $n_k \in src(e_k)$, $n_{k+1} = trg(e_k)$ for $k \in \{0, \ldots, m-1\}$. The *length* of this path is m, i.e., the number of traversed edges; if $m = 0$, the path is *empty*. A *cycle* is a path like above where $n_0 = n_m$.

Definition 9 (directed acyclic graphs, dags). *A* directed acyclic graph *or* dag (over Σ) *is a labelled graph which does not contain any non-empty cycle.*

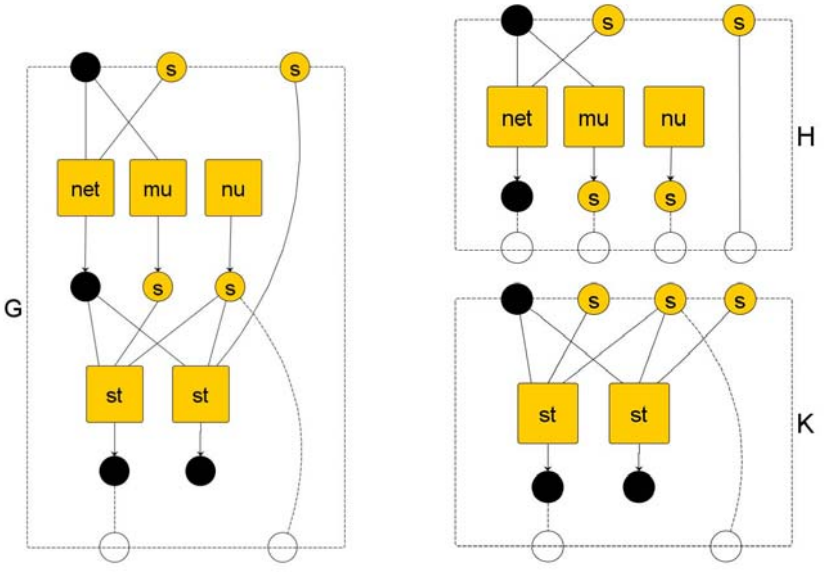


Fig. 5. Some sample term graphs

In the next definition we equip dags with some *attaching points* or *interfaces*, which will be used later to define suitable operations on them. We assume that an arbitrary but fixed signature Σ over a set of sorts S is given.

Definition 10 (ranked dags). *A* (u, w)-ranked dag*, with* $u, w \in S^*$ *(a* dag of rank (u, w)*) is a triple* $g = \langle v, d, r \rangle$*, where* $d = \langle N, E, l_N, l_E, src, trg \rangle$ *is a dag with exactly* $\#u$ *empty nodes,* $v \colon \underline{\#u} \to N_\emptyset$ *is a bijection between* $\underline{\#u}$ *and the empty nodes of* d*, called the* variable mapping*, and satisfying* $l_N(v(j)) = u|_j$ *for all* $j \in \underline{\#u}$*, and* $r \colon \underline{\#w} \to N$ *is a function, called the* root mapping*, satisfying* $l_N(r(i)) = w|_i$ *for all* $i \in \underline{\#w}$*.*

Two (u, w)-ranked dags $g = \langle v, d, r \rangle$ and $g' = \langle v', d', r' \rangle$ are *isomorphic* if there exists a *ranked dag isomorphism* $\phi : g \to g'$ between them, i.e., a pair of bijections $\phi_N \colon N_d \to N_{d'}$ and $\phi_E \colon E_d \to E_{d'}$ preserving connections, labels, roots and variables in the expected way.

Definition 11 (ranked term graphs). *A* (u, w)-ranked term graph *is an isomorphism class* $T = [g]$ *of* (u, w)*-ranked dags. We write* T^u_w *to recall that* T *has rank* (u, w)*.*

Figure 5 illustrates the graphical conventions we use by showing three term graphs over the set of sorts $S = \{\bullet, \mathsf{s}\}$ and the signature $\Sigma = \{\mathsf{nu} \colon \epsilon \to \mathsf{s}, \mathsf{mu} \colon \bullet \to \mathsf{s}, \mathsf{net} \colon \bullet\,\mathsf{s} \to \bullet, \mathsf{st} : \bullet\mathsf{ss} \to \bullet\}$. Nodes are depicted as small circles and edges as rounded boxes, each with the corresponding label inside, but for $\bullet$-labeled nodes which are drawn as black circles. For each edge, the nodes in its source connection are linked with plain lines coming from above and they

are ordered from left to right, while a down-going arrow connects an edge to its target node. Clearly, the connection functions are consistent with the labelling; for example each edge labeled by st is linked from above to three nodes labeled •, s and s, in this order. By condition 2 of Definition 8 every node has at most one incoming arrow.

The outer dashed rounded boxes conceptually represent the interfaces of the term graphs. The top border, on which all the empty nodes are placed, encodes the variable mapping: an empty node m is in the i-th position (from left to right) if and only if $v(i) = m$. The bottom border depicts instead the root mapping: each "fake" node on it represents an index, and it is connected with a dashed line to its image. Therefore the three term graphs G, H and K have rank (•ss, •s), (•ss, •sss) and (•sss, •s), respectively.

It is fair to notice that these graphical conventions are not standard: in other papers the direction of arrows is reversed, and/or the drawing is flipped vertically. Our choice is consistent with a data-flow interpretation of such graphs, where data flows from top to bottom: every node represents a value that is either produced along the only arrow pointing to it, or that will become available from the environment if the node is empty (and thus it is a variable). Each value "stored" within a node can be used several times along the (possibly dashed) lines that leave downwards. Each edge processes the inputs coming from above and produces one result in its target node. The data-flow orientation is the most appropriate one for presenting the encoding of algebra **AGN** in Section 4, following the intuitive drawing of hierarchical structures.

We introduce now two operations on term graphs. The *composition* of two ranked term graphs is obtained by gluing the variables of the first one with the roots of the second one, and it is defined only if they correspond in number and sorts. The *union* of term graphs instead is always defined, and it is a kind of disjoint union where roots and variables are suitably concatenated.

Definition 12 (composition and union of ranked term graphs).
Composition. *Let* $T^u_w = [\langle v, d, r\rangle]$ *and* $T'^w_z = [\langle v', d', r'\rangle]$ *be ranked term graphs. Their* composition *is the ranked term graph* $S^u_z = T^u_w ; T'^w_z$ *defined as* $[\langle in_d \circ v, d'', in_{d'} \circ r'\rangle]$, *where* d'' *is obtained from* $d \uplus d'$, *the disjoint union of* d *and* d' *(component-wise on edges and on nodes), modulo the least equivalence relation such that* $r(i) = v'(i)$ *for all* $i \in \underline{\#w}$ *(i.e., by identifying the* i*-th root of* T *with the* i*-th variable of* T'*), and* $in_d, in_{d'}$ *are the inclusions of* d, d' *into* d''.

Union. *Let* $T^u_w = [\langle v, d, r\rangle]$ *and* $T'^x_y = [\langle v', d', r'\rangle]$ *be ranked term graphs. Their* union *or* parallel composition *is the ranked term graph* $S^{ux}_{wy} = T^u_w \otimes T'^x_y$ *defined as* $[\langle v'', d \uplus d', r''\rangle]$, *where* $v'' : \underline{\#(ux)} \to N_\emptyset \uplus N'_\emptyset$ *is defined as* $v''(i) = v(i)$ *if* $i \in \underline{\#u}$, *and* $v''(i) = v'(i - \#u)$ *if* $i \in \{\#u + 1, \ldots, \#(ux)\}$; *and* $r'' : \underline{\#(wy)} \to N \uplus N'$ *is defined similarly.*

Figure 5 depicts $G = H ; K$, i.e., the term graph $G^{\bullet ss}_{\bullet s}$ is the composition of $H^{\bullet ss}_{\bullet sss}$ and $K^{\bullet sss}_{\bullet s}$. The operations of composition and union on ranked term graphs satisfy various algebraic laws, but we refrain from listing them here because

$$\text{(op)}\ \frac{f \in \Sigma_{u,s}}{f : u \to s} \qquad \text{(id)}\ \frac{u \in S^*}{id_u : u \to u} \qquad \text{(bang)}\ \frac{u \in S^*}{!_u : u \to \epsilon} \qquad \text{(dup)}\ \frac{u \in S^*}{\nabla_u : u \to uu}$$

$$\text{(sym)}\ \frac{u, v \in S^*}{\rho_{u,v} : uv \to vu} \qquad \text{(seq)}\ \frac{t : u \to v \quad t' : v \to w}{t; t' : u \to w} \qquad \text{(par)}\ \frac{t : u \to v \quad t' : u' \to v'}{t \otimes t' : uu' \to vv'}$$

Fig. 6. Inference rules of gs-monoidal theories

they will follow from the result reported in the next section, showing that term graphs form the initial model of gs-monoidal theories (see Theorem 2).

3.1 GS-Monoidal Theories

As anticipated in the Introduction, inspired by the seminal work on flownomial algebras in [12], a sound and complete axiomatisation of ranked term graphs has been proposed in [10]. This result is analogous to the characterisation of (tuples of) terms over a signature Σ as arrows of the Lawvere theory of Σ, considered as the free cartesian category generated by Σ.

However, the categorical framework where such results have been proved is not relevant here, because we are not interested in the details of the proofs, but just in the axiomatisation itself, which allows us to represent every ranked term graph as an expression using suitable operators. The properties of such operators are described by a set of axioms, and the main fact is that equivalence classes of expressions with respect to the axioms are one-to-one with ranked term graphs.

The expressions of interest are generated by the rules depicted in Fig. 6: they are obtained from some basic (families of) terms by closing them with respect to *sequential* (seq) and *parallel* (par) *composition*. By rule (op), the basic terms include one generator for each operator of the signature: these are the elementary bricks of our expressions, and conceptually correspond to the hyper-edges of the term graphs. All other basic terms define the wires that can be used to build our graphs: the *identities* (id), the *dischargers* (bang), the *duplicators* (dup) and the *symmetries* (sym). Every expression $t\colon u \to v$ generated by the inference rules is typed by a *source* and by a *target* sequence of sorts (u and v, respectively), which are relevant only for the sequential composition, which is a partial operation. The next definition presents the axioms imposed on such expressions.

Definition 13 (gs-monoidal theory). *Given a signature Σ over a set of sorts S, the* gs-monoidal theory $\mathbf{GS}(\Sigma)$ *is the category whose objects are the elements of S^* and whose arrows are equivalence classes of* gs-monoidal terms*, i.e., terms generated by the inference rules in Fig. 6 subject to the following conditions*

- *identities and sequential composition satisfy the axioms of categories*
  [identity] $id_u \,;\, t = t = t \,;\, id_v$ *for all* $t : u \to v$*;*
  [associativity] $t_1 \,;\, (t_2 \,;\, t_3) = (t_1 \,;\, t_2) \,;\, t_3$ *whenever any side is defined,*

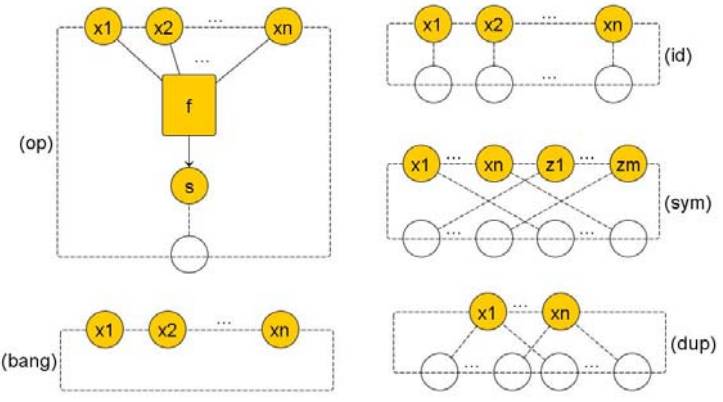


Fig. 7. The term graphs corresponding to the basic arrows of the gs-monoidal theory

- $\otimes$ *is a monoidal functor with unit* id_ϵ*, i.e., it satisfies*
  [functoriality] $id_{uv} = id_u \otimes id_v$*, and*
  $(t_1 \otimes t_2)\ ;\ (t'_1 \otimes t'_2) = (t_1\ ;\ t'_1) \otimes (t_2\ ;\ t'_2)$ *whenever both sides are defined,*
  [monoid] $t \otimes id_\epsilon = t = id_\epsilon \otimes t \qquad t_1 \otimes (t_2 \otimes t_3) = (t_1 \otimes t_2) \otimes t_3$
- ρ *is a symmetric monoidal natural transformation, i.e., it satisfies*
  [naturality] $(t \otimes t')\ ;\ \rho_{v,v'} = \rho_{u,u'}\ ;\ (t' \otimes t)$ *for all* $t : u \to v$ *and* $t' : u' \to v'$
  [symmetry] $(id_u \otimes \rho_{v,w})\ ;\ (\rho_{u,w} \otimes id_v) = \rho_{u \otimes v,w} \qquad \rho_{u,v}\ ;\ \rho_{v,u} = id_{u \otimes v}$
  $\rho_{\epsilon,u} = \rho_{u,\epsilon} = id_u$
- ∇ *and* $!$ *satisfy the following axioms*
  [unit] $!_\epsilon = \nabla_\epsilon = id_\epsilon$
  [duplication] $\nabla_u\ ;\ (id_u \otimes \nabla_u) = \nabla_u\ ;\ (\nabla_u \otimes id_u) \qquad \nabla_u\ ;\ (id_u \otimes !_u) = id_u$
  $\nabla_u\ ;\ \rho_{u,u} = \nabla_u$
  [monoidality] $\nabla_{uv}\ ;\ (id_u \otimes \rho_{v,u} \otimes id_v) = \nabla_u \otimes \nabla_v \qquad !_{uv} = !_u \otimes !_v$

A wiring *is an arrow of* $\mathbf{GS}(\Sigma)$ *which is obtained from the rules of Fig. 6 without using rule (op).*

Notice that the definition of wiring is well-given, because any operator symbol introduced by rule (op) is preserved by all the axioms of the theory.

Given the above definition, the main result of [10] is summarised as follows.

Theorem 2 (axiomatisation of ranked term graphs [10]). *Let* Σ *be a signature over a set of sorts* S *and let* $u, v \in S^*$*. Then there is a bijective correspondence between term graphs over* Σ *of rank* (u, v) *and arrows of the gs-monoidal theory of* Σ*,* $\mathbf{GS}(\Sigma)$*, from* u *to* v*. In particular, wirings from* u *to* v *are in bijective correspondence with discrete term graphs of rank* (u, v)*.*

Just to give a feeling on how the correspondence stated by the theorem works, the term graph denoted by a gs-monoidal term generated by the rules of Fig. 6 can be built by structural induction: Fig. 7 shows the ranked term graphs corresponding to the basic terms introduced by rules (op), (id), (bang), (dup) and (sym),

assuming that $u = x_1, x_2, \cdots, x_n$ and $v = z_1, z_2, \cdots, z_m$; instead, rules (seq) and (par) correspond to the operations of composition and of union as introduced in Definition 12. For example, the (equivalence classes of) terms $t_H \triangleq [((\nabla_\bullet \otimes id_s)\ ;\ (id_\bullet \otimes \rho_{\bullet,s})\ ;\ (\mathsf{net} \otimes \mathsf{mu} \otimes \mathsf{nu})) \otimes id_s] \colon \bullet\mathsf{ss} \to \bullet\mathsf{sss}$ and $t_K \triangleq [(((\nabla_\bullet \otimes id_s \otimes \nabla_s)\ ;\ (id_\bullet \otimes \rho_{\bullet,ss} \otimes \nabla_s)) \otimes id_s)\ ;\ (\mathsf{st} \otimes ((id_{\bullet s} \otimes \rho_{s,s})\ ;\ ((\mathsf{st}\ ;!_\bullet) \otimes id_s)))] \colon \bullet\,\mathsf{sss} \to \bullet\mathsf{s}$ denote, respectively, the term graphs H and K from Fig. 5, while G corresponds to $t_H\ ;\ t_K$.

In the following we shall assume that $\otimes$ has precedence over ;, hence in the example above we can write for example $t_H \triangleq [(\nabla_\bullet \otimes id_s\ ;\ id_\bullet \otimes \rho_{\bullet,s}\ ;\ \mathsf{net} \otimes \mathsf{mu} \otimes \mathsf{nu}) \otimes id_s] \colon \bullet\mathsf{ss} \to \bullet\mathsf{sss}$, omitting several brackets.

An easy corollary of Theorem 2, that we will need later on, states that each wiring of $\mathbf{GS}(\Sigma)$ denotes a suitable sort-preserving function.

Corollary 1. *Let $u, v \in S^*$. Then the wirings of $\mathbf{GS}(\Sigma)$ from u to v are in bijective correspondence with the set of functions $\{k \colon \underline{\#v} \to \underline{\#u} \mid u|_{k(i)} = v|_i$ for all $i \in \underline{\#v}\}$.*

In fact, each wiring from u to v denotes a discrete term graph of rank (u, v), which is uniquely determined by its root function.

4 From AGN Terms to Term Graphs

In this section we define a translation from the terms of the algebra **AGN** introduced in Section 2 to equivalence classes of terms of the gs-monoidal theory of a suitable signature, and therefore, by Theorem 2, to term graphs over that signature. Disregarding for the moment the technical details, the intuition behind the translation is quite simple: the nesting of edges is rendered in a term graph by a tree-like structure made of nodes of a special sort, representing locations; free nodes are mapped to variables; and each restricted node is encoded as the target node of a special constant.

As a first step, we introduce the signature over which the term graphs obtained by the translation of the terms of **AGN** are defined.

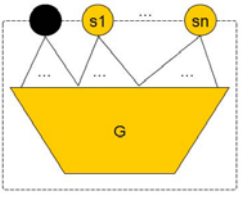
Definition 14 (signature $\Sigma_B^\bullet$). *Given the set of sorts $S^\bullet = S \cup \{\bullet\}$, assuming that $\bullet \notin S$, the signature $\Sigma_B^\bullet$ over $S^\bullet$ is defined as follows*

$$\Sigma_B^\bullet = \{b \colon \bullet, rnk(b) \to \bullet \mid b \in B\} \cup \{\nu_s \colon \epsilon \to s, \mu_s \colon \bullet \to s \mid s \in S\}$$

Intuitively, each box label $b \in B$ corresponds to an operator symbol, having as arity the rank of b preceded by $\bullet$, and $\bullet$ as coarity. The new sort $\bullet$ has a special role in our encoding, because it represents the *locations* in a graph with nesting. An edge labelled with the operator b will be connected (by its first source) to the node representing the location where it lies, and it will "offer" a new location (the target connection), conceptually corresponding to its interior. Furthermore, the signature includes the operator symbols ν_s and μ_s for each sort s: these will be connected through their target connection to the node they restrict; μ_s

additionally has one source of type $\bullet$, which matches with the intuitive definition of "localised restriction".

A term $\mathbb{G}$ of the algebra **AGN** has a *set* of free nodes $fn(\mathbb{G})$ which are used as an interface to the environment, as seen in Section 2. Instead, the "interface" of a term graph is a pair of *lists* of sorts, forming its rank. Each term $\mathbb{G}$ will be translated to a term graph having an empty list of roots, and a linearisation of the free nodes of $\mathbb{G}$ as variables (as exemplified in the figure on the right): therefore the translation is parametric w.r.t. an *assignment*, i.e., a function which assigns a positional index to each free node of the term.



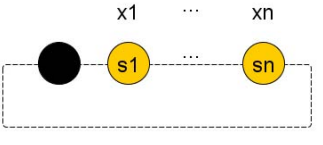
A generic term $\mathbb{G}$ of **AGN** as a term graph (with $fn(\mathbb{G}) = \{x_1 : s_1, \cdots, x_n : s_n\}$)

Definition 15 (Assignment). *An* assignment *is a function* $\sigma \in \bigcup_{n\in\mathbb{N}}\{f : \underline{n} \to \mathcal{X}\times S \mid f \text{ is injective}\}$. *An assignment* $\sigma : \underline{n} \to \mathcal{X}\times S$ *for a given* $n \in \mathbb{N}$ *is uniquely determined by a list of nodes without repetitions (because it is injective), namely* $\sigma(1), \sigma(2), \ldots, \sigma(n)$*: we shall often represent it this way and write* $x : s \in \sigma$ *as a shorthand for* $x : s \in img(\sigma)$*, the image of* σ*.*

In the following, by $\tau(\sigma)$ we denote $\tau(\sigma(1), \sigma(2), \ldots, \sigma(n))$, i.e., the sequence of sorts of the nodes in $img(\sigma)$. Furthermore, for a given list of nodes $\overline{y} \in (\mathcal{X} \times S)^*$ and an assignment σ such that $|\overline{y}| \subseteq img(\sigma)$, we let $k^{\sigma}_{\overline{y}} : \underline{\#\overline{y}} \to \underline{\#\sigma}$ be the function such that $k^{\sigma}_{\overline{y}}(i) = \sigma^{-1}(\overline{y}|_i)$ for all $i \in \underline{\#\overline{y}}$.

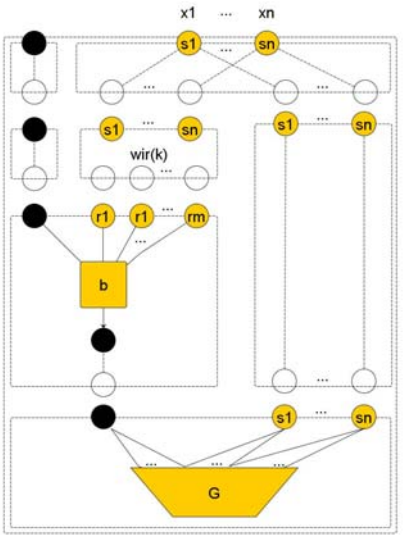
Definition 16 (Encoding AGN into gs-monoidal terms). *Let* $\mathbb{G}$ *be a term of* **AGN** *over sorts* S*, box labels* B *and names* $\mathcal{X}$*, and let* $\sigma = x_1 : s_1, \ldots, x_n : s_n$ *be an assignment. We say that* $[\![\mathbb{G}]\!]_\sigma$ *is* well-defined *if* $fn(\mathbb{G}) \subseteq img(\sigma)$*; in this case,* $[\![\mathbb{G}]\!]_\sigma$ *is a term graph of rank* $((\bullet, \tau(\sigma)), \epsilon)$ *over the signature* $\Sigma^{\bullet}_B$*, defined by structural induction as follows (recall that* $\otimes$ *has precedence over* $;$*)*

- $[\![\mathbf{0}]\!]_\sigma = [!_{\bullet,\tau(\sigma)}] : \bullet, \tau(\sigma) \to \epsilon$
- $[\![x : s]\!]_\sigma = [!_{\bullet,\tau(\sigma)}] : \bullet, \tau(\sigma) \to \epsilon$

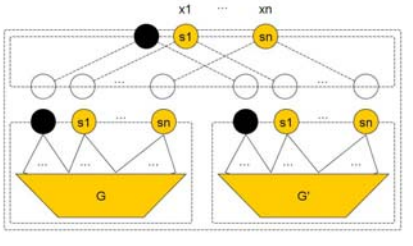


The encodings $[\![0]\!]_\sigma$ *and* $[\![x_i : s_i]\!]_\sigma$*, graphically, assuming* $\sigma = x_1 : s_1, \ldots, x_n : s_n$ *and* $i \in \underline{n}$*.*

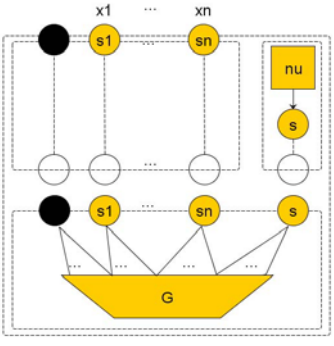
- $[\![b[\mathbb{G}](\overline{y})]\!]_\sigma = [id_\bullet \otimes \nabla_{\tau(\sigma)} \; ; \; (id_\bullet \otimes wir(k) \; ; \; b) \otimes id_{\tau(\sigma)}] \; ; \; [\![\mathbb{G}]\!]_\sigma : \bullet, \tau(\sigma) \to \epsilon$, *where the gs-term* $wir(k) : \tau(\sigma) \to rnk(b)$ *is any representative of the wiring uniquely determined (according to Corollary 1) by the function* $k \triangleq k^{\sigma}_{\overline{y}} : \underline{\#rnk(b)} \to \underline{\#\sigma}$ *defined above (see also Fig. 8)*
- $[\![\mathbb{G}|\mathbb{G}']\!]_\sigma = [\nabla_{\bullet,\tau(\sigma)}] \; ; \; [\![\mathbb{G}]\!]_\sigma \otimes [\![\mathbb{G}']\!]_\sigma : \bullet, \tau(\sigma) \to \epsilon$



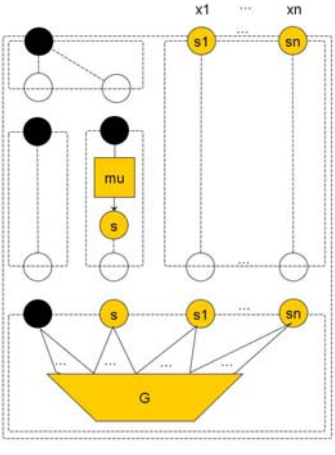
$[\![b[\mathbb{G}](\overline{y})]\!]_\sigma$



$[\![\mathbb{G}|\mathbb{G}']\!]_\sigma$



$[\![(\nu\ x:s)\mathbb{G}]\!]_\sigma$



$[\![(\mu\ x:s)\mathbb{G}]\!]_\sigma$

Fig. 8. The encoding of the terms of algebra **AGN**, graphically

– $[\![(\nu\ x:s)\mathbb{G}]\!]_\sigma = [id_{\bullet,\tau(\sigma)} \otimes \nu_s]\ ;\ [\![\mathbb{G}\{^{y:s}/_{x:s}\}]\!]_{\sigma,y:s} : \bullet, \tau(\sigma) \to \epsilon$
where $y:s = fresh_{\mathbb{G}}(x:s,\sigma)$

– $[\![(\mu\ x:s)\mathbb{G}]\!]_\sigma = [(\nabla_\bullet\ ;\ id_\bullet \otimes \mu_s) \otimes id_{\tau(\sigma)}]\ ;\ [\![\mathbb{G}\{^{y:s}/_{x:s}\}]\!]_{y:s,\sigma} : \bullet, \tau(\sigma) \to \epsilon$
where $y:s = fresh_{\mathbb{G}}(x:s,\sigma)$

In the last two rules, $fresh_{\mathbb{G}}(x{:}s,\sigma)$ is a function returning $x{:}s$ itself if $x{:}s \notin \sigma$, and returning a fresh s-sorted node (not appearing neither in σ nor in $\mathbb{G}$) otherwise. Notice that $[\![\mathbf{0}]\!]_\sigma$ and $[\![x:s]\!]_\sigma$ are defined in the same way, but the first is defined for any σ, while the second one is defined only if $x:s \in \sigma$.

Our first main result shows that the encoding is sound w.r.t. the equivalence $\equiv_{\mathcal{A}}$, i.e., that $\equiv_{\mathcal{A}}$-equivalent **AGN** terms are mapped to the same term graph.

Theorem 3 (soundness). *Let* $\mathbb{G}$ *and* $\mathbb{H}$ *be two terms of algebra* **AGN** *over sorts* S*, box labels* B *and names* $\mathcal{X}$ *such that* $\mathbb{G} \equiv_{\mathcal{A}} \mathbb{H}$*. Then, for any assignment* σ*: 1)* $[\![\mathbb{G}]\!]_\sigma$ *is well-defined iff* $[\![\mathbb{H}]\!]_\sigma$ *is such; 2)* $[\![\mathbb{G}]\!]_\sigma = [\![\mathbb{H}]\!]_\sigma$ *when well-defined.*

Proof (sketch). Item 1) follows by the fact that $\mathbb{G} \equiv_{\mathcal{A}} \mathbb{H}$ implies $fn(\mathbb{G}) = fn(\mathbb{H})$.

To prove item 2), one should show that each axiom A1–A10 from Definition 5 is preserved by the encoding, i.e., that the encoding of the left-hand side of each axiom can be proved equal to the encoding of the right-hand side by exploiting the axioms of gs-monoidal theories (see Definition 13). Note that for axioms A4–A7 one has to consider separately the cases for ν and μ. The detailed proof is omitted for space constraints. □

The second main result of this section states that the encoding is complete w.r.t. the equivalence $\equiv_{\mathcal{A}}$, i.e., that any two **AGN** terms mapped to the same term graph must be $\equiv_{\mathcal{A}}$-equivalent.

Theorem 4 (completeness). *Let* $\mathbb{G}$ *and* $\mathbb{H}$ *be two terms of algebra* **AGN** *over node sorts* S*, box labels* B *and variables* $\mathcal{X}$*. If for all assignments* σ *it holds* $[\![\mathbb{G}]\!]_\sigma = [\![\mathbb{H}]\!]_\sigma$*, then* $\mathbb{G} \equiv_{\mathcal{A}} \mathbb{H}$*.*

Proof (sketch). Let us assume, without loss of generality, that terms $\mathbb{G}$ and $\mathbb{H}$ are in normalised form, and that for all assignments σ it holds $[\![\mathbb{G}]\!]_\sigma = [\![\mathbb{H}]\!]_\sigma$. Then they must have the same free nodes, because if $fn(\mathbb{G}) \neq fn(\mathbb{H})$, then it is immediate to find a σ for which only one between $[\![\mathbb{G}]\!]_\sigma$ and $[\![\mathbb{H}]\!]_\sigma$ is defined, contradicting the hypothesis. Next, taken a generic σ such that $fn(\mathbb{G}) \subseteq img(\sigma)$, it can be shown that the rules for the encoding $[\![\_]\!]_\sigma$ induce a suitable partial bijection between the nodes of the syntax tree of $\mathbb{G}$ and the edges of the term graph $[\![\mathbb{G}]\!]_\sigma$ (see also Fig. 8). Since $[\![\mathbb{G}]\!]_\sigma = [\![\mathbb{H}]\!]_\sigma$, these partial bijections can be composed obtaining a partial bijection between the nodes of $\mathbb{G}$ and those of $\mathbb{H}$, which allows us to conclude that they are $\equiv_{\mathcal{A}}$-equivalent, by the considerations at the end of Section 2.3. □

We conclude this section by showing in Fig. 9 the term graph $[\![\mathbb{G}_G]\!]_\sigma$, obtained by applying the encoding of Definition 16 to the NR-graph of Fig. 1 (see Example 2 for the defining expression of $\mathbb{G}_G$), with substitution $\sigma = \{x : s\}$. Because of layout constraints, the term graph is rotated counter-clockwise, exposing the variable mapping on the left border.

5 From Term Graphs to Graphs with Nesting

In this section we prove that there is a one-to-one correspondence between the term graphs obtained by encoding terms of the algebra **AGN** and the NR-graphs introduced in Section 2: this will conclude the proof of Theorem 1.

The first point we address is whether or not the encoding presented in the previous section is surjective, i.e., if any term graph in $\mathbf{GS}(\Sigma_B^\bullet)$ is the image of some term of the algebra **AGN** (for some σ). As our encoding maps to term graphs of rank $(\bullet u, \epsilon)$ only (with $u \in S^*$), the answer is clearly negative in

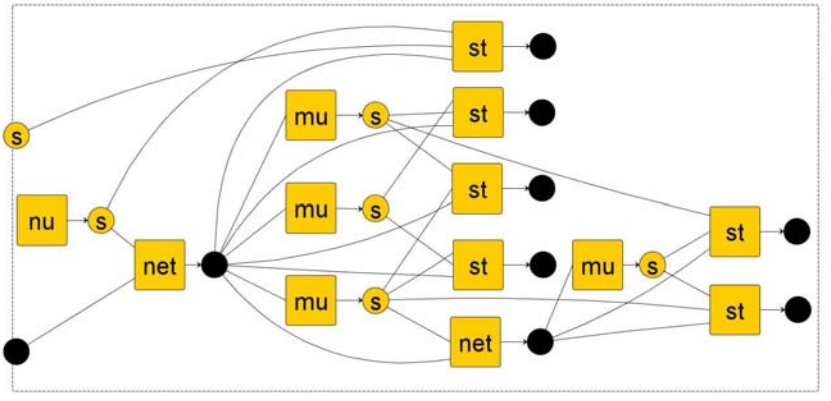
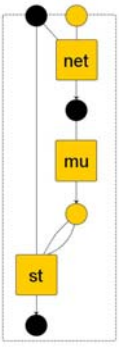


Fig. 9. The term graph $[\![\mathbb{G}_G]\!]_\sigma$ (see Example 2 and Fig. 1)

general. However, even if we restrict to consider term graphs of rank $(\bullet u, \epsilon)$, where $u \in S^*$, the mapping is not surjective. The crucial fact is that the scoping discipline of μ-restriction restricts the visibility of a locally restricted node $x : s$ in such a way that it cannot be used from edges outside the one where $(\mu\, x : s)$ appears, but such a node scoping discipline has no counterpart in term graphs.



Example 4. Let us consider the algebra for our running example of network systems. Then for the term graph $t \triangleq [\nabla_\bullet \otimes id_s \,;\, id_\bullet \otimes (\mathsf{net} \,;\, \mu_s \,;\, \nabla_s) \,;\, \mathsf{st} \,;\, !_\bullet] \colon \bullet\, \mathsf{s} \to \epsilon$ (see the figure to the right) there is no **AGN** term $\mathbb{G}$ such that $[\![\mathbb{G}]\!]_{x:\mathsf{s}} = t$. In fact, among the natural candidates: the term $\mathsf{net}[(\mu\, y)\mathsf{st}(y, y)](x)$ would be encoded as $[\mathsf{net} \,;\, \nabla_\bullet \,;\, id_\bullet \otimes (\mu_s \,;\, \nabla_s) \,;\, \mathsf{st} \,;\, !_\bullet]$ with st lying "under" net (and not being a "sibling" of net like in t); the term $\mathsf{net}(x)|(\mu\, y)\mathsf{st}(y, y)$ would be encoded as $[\nabla_\bullet \otimes id_s \,;\, (\nabla_\bullet \,;\, id_\bullet \otimes (\mu_s \,;\, \nabla_s) \,;\, \mathsf{st} \,;\, !_\bullet) \otimes (\mathsf{net} \,;\, !_\bullet)]$ with net and st siblings, but the restriction appearing "outside" net (and not "inside" net, as in t); the term $\mathsf{net}[(\mu\, y)\mathbf{0}](x)|\mathsf{st}(y)$ would have $y : s$ as a free node.

The above counterexample suggests that the algebra **AGN** can serve to characterise exactly those term graphs with well-scoped references to nodes. These are defined as follows.

Definition 17 (well-scoped term graphs). *Let $T = [\langle v, d, r\rangle]$ be a term graph of rank $((\bullet, \tau(\sigma)), \epsilon)$ over the signature $\Sigma_B^\bullet$, with $d = \langle N, E, l_N, l_E, src, trg\rangle$. We say that T is* well-scoped *if for all $e \in E$, for all $n \in src(e)$, if there exists $e' \in E$ such that $n = trg(e')$ and $l_E(e') = \mu$, then $src(e')$ is on a $\bullet$-path (i.e., a path where all nodes are of sort $\bullet$) from $src(e)|_1$ to $v(1)$.*

Informally, this means that in a well-scoped term graph, every edge referring to a locally restricted node n must lie inside the location where n is restricted.

Proposition 2 (AGN terms and well-scoped term graphs). *Given a set of sorts S, a set of ranked labels B and a set of variables $\mathcal{X}$, for each assignment $\sigma = x_1 : s_1, \ldots, x_n : s_n$ there is a one-to-one correspondence between the equivalence classes w.r.t. $\equiv_{\mathcal{A}}$ of* **AGN** *terms with free names in $\{x_1, \ldots, x_n\}$ and well-scoped term graphs of rank $((\bullet, \tau(\sigma)), \epsilon)$ over signature $\Sigma_B^{\bullet}$.*

Proof (sketch). By structural induction it is possible to show that the result of the encoding of Definition 16 is a gs-monoidal term corresponding to a well-scoped term graph; furthermore, by structural induction on the gs-monoidal normal form of well-scoped term graphs, it can be shown that every well-scoped term graph can be obtained as the result of the encoding of a suitable **AGN** term. By Theorem 3 the encoding is consistent with $\equiv_{\mathcal{A}}$-equivalence classes, and by Theorem 4 it is injective on term graphs. □

Well-scoped term graphs can be considered just as an alternative, graphical representation of NR-graphs, where the nesting is represented by a tree of locations, i.e., the $\bullet$-sorted nodes. Formally, this relationship is captured by the next definition and the following result.

Definition 18 (from term graphs to NR-graphs). *Let $T = [\langle v, d, r\rangle]$ be a term graph of rank $((\bullet, \tau(\sigma)), \epsilon)$ over signature $\Sigma_B^{\bullet}$, with $d = \langle N, E, l_N, l_E, src, trg\rangle$.*

For a $\bullet$-sorted node $n \in N$, let $\mathcal{NR}_G(n)$ be the NR-graph defined as $\mathcal{NR}_G(n) = \langle LR, D, l, c, \rho\rangle$, with

- $LR = \{trg(e) : s \mid e \in E \wedge src(e)|_1 = n \wedge l_E(e) = \mu_s\}$
- $D = \{e \in E \mid src(e)|_1 = n \wedge l_E(e) \in B\}$
- $l(e) = l_E(e)$ *for all* $e \in D$
- $c(e) = u$ *when* $src(e) = \bullet u$, *for all* $e \in D$
- $\rho(e) = \mathcal{NR}_G(trg(e))$ *for all* $e \in D$.

Furthermore, let $\mathcal{NR}(T) = \langle FN, GR, \mathcal{NR}_G(v(1))\rangle$, with

- $FN = \{v(i) : l_N(v(i)) \mid 1 < i \leq \#\tau(\sigma)\}$
- $GR = \{trg(e) : s \mid e \in E \wedge l_E(e) = \nu_s\}$.

Proposition 3 (correctness of the encoding). *In the hypotheses of Definition 18, $\mathcal{NR}(T)$ is a NR-graph if and only if T is well-scoped. Furthermore, the encoding does not depend on the choice of $\langle v, d, r\rangle$ in the equivalence class T (in the sense that the same NR-graph is obtained, up to isomorphism).*

Proof (sketch). By Definition 1 the main fact to prove is that in every NR-graph inside $\mathcal{NR}(T)$ the edges are connected only to available external nodes, i.e., either to global nodes, or to those locally restricted in an enclosing edge. But this is exactly the property ensured by being well-scoped. □

An encoding in the opposite direction, from NR-graphs to well-scoped term graphs, can be defined as well, but it requires more care. In fact, the naive approach of "flattening" the nested structure of the NR-graph by rearranging all its nodes and edges in a single structure might not work, because locally

restricted nodes or edges in different sub-graphs could have the same identity. Therefore starting with an NR-graph G, one should first obtain an isomorphic G' with all node and edge identities distinct, and then one can proceed with the flattening. We do not present here the technical details of this construction, but we state its existence, and that it is inverse to $\mathcal{NR}$.

Proposition 4 (from NR-graphs to term graphs). *There exists an encoding $\mathcal{TG}$ such that if σ is an assignment and G is an NR-graph with free nodes in $img(\sigma)$, then $\mathcal{TG}(G,\sigma)$ is a well-scoped term graph of rank $((\bullet, \tau(\sigma)), \epsilon)$. Furthermore, for each well-scoped term graph T of rank $((\bullet, \tau(\sigma)), \epsilon)$, $\mathcal{TG}(\mathcal{NR}(T), \sigma) = T$, and for each NR-graph G with free nodes in $img(\sigma)$, $\mathcal{NR}(\mathcal{TG}(G,\sigma)) \cong G$.*

Note that the equality is strict in $\mathcal{TG}(\mathcal{NR}(T), \sigma) = T$, because the NR-graph $\mathcal{NR}(T)$ obtained from T has all nodes and edges distinct by construction, whereas the equality is only up to isomorphism in $\mathcal{NR}(\mathcal{TG}(G,\sigma)) \cong G$ because $\mathcal{TG}$ may involve the renaming of some nodes and edges.

6 Towards an Enhanced Modelling Framework

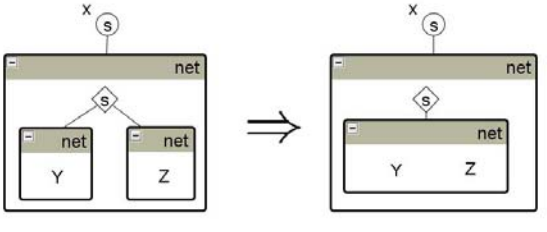
In the previous sections we presented the main technical results, which can be summarised by the following diagram, already presented in the introduction.

$$\begin{array}{ccc} \mathbf{AGN}(S,B)/_{\equiv_{\mathcal{A}}} & \Longleftrightarrow & \text{NR-Graphs over } (S,B) \\ \Big\downarrow \text{Sec. 4} & & \Big\downarrow \text{Sec. 5} \\ \mathbf{GS}(\Sigma_B^\bullet) & \xleftarrow{\quad [10] \quad} & \text{Term Graphs over } \Sigma_B^\bullet \end{array}$$

Therefore we established a one-to-one correspondence between **AGN** terms up to equivalence, and well-scoped term graphs and NR-graphs up to isomorphism. In this section we first discuss how the relationship with term graphs can be exploited to enrich the visual framework of NR-graphs with a notion of rewriting and with existing analysis techniques. Next we discuss possible generalisations of the proposed framework

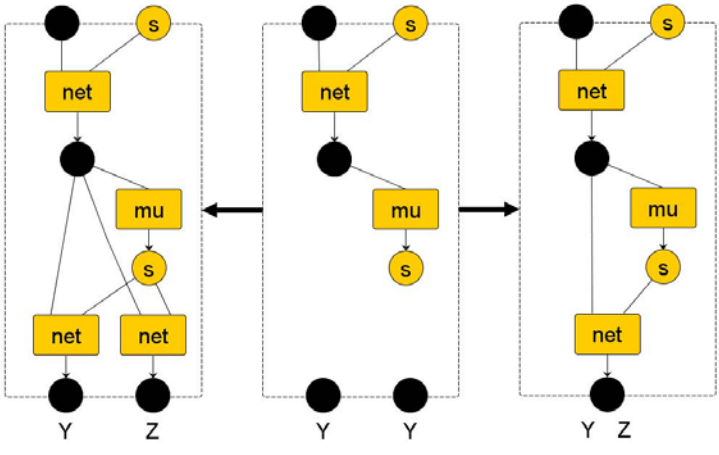
6.1 On Rewriting NR-Graphs

In order to equip our models with behavioural specifications, a natural way is to look for a suitable notion of graph transformation over NR-graphs. For example, one could consider the following transformation rule, intended to model the fact that two local sub-networks can be merged into a single one, which will include the contents of both.



Clearly, one should formalise this notion of rule, as well as its operational meaning, i.e., when it can be applied to a given NR-graph and what the result is. For example, one should clarify the meaning and the role of the variables Y and Z, which are intended to denote the whole content of an edge.

The one-to-one correspondence with term graphs, for which a notion of rewriting is well-understood, can be helpful in this respect. For example, we can translate the left- and the right-hand side of the rule into term graphs, and we can introduce a third term graph in the middle and two morphisms in order to relate the items that have to be preserved, obtaining the *double-pushout* rule



Quite naturally, through this translation the variables correspond to •-labelled nodes, i.e., to locations. This encoding of NR-graph rules into term graph rules can be exploited directly by lifting the definition of term graph rewriting to NR-graphs, or can be used to check the consistency of an original notion of rewriting over NR-graphs. In both cases, it provides a direct link to the rich theory of concurrency and parallelism developed for the algebraic approaches to graph transformation, as well as to the verification techniques developed for them [1]: how far these results can be applied to NR-graphs and their transformations is a subject of future work.

6.2 Edges with Inner Rank: From Term Graphs to GS-Graphs

The distinguishing feature of NR-graphs is the fact that edges are regarded as containers of nested subgraphs. A natural generalisation of this idea would be to equip each edge also with a sorted *inner interface*: while the ordinary "outer interface" is induced by the nodes where the box is attached to, the inner interface would introduce a dual view of local nodes provided by the edge to the nested graph. Such an inner interface could be partly modelled using μ-restriction, but with two main differences: 1) the sorting of the inner interface would be fixed at the signature level, while μ-restricted nodes of any sort can always occur inside a box; 2) the order of nodes provided by the inner interface would be

fixed, while μ-restrictions can commute thanks to axiom A4. For example, by equipping each edge with an inner interface having the same rank of the outer interface would provide a straight modelling of modules (each edge) with formal parameters (the nodes provided by the inner interface) and actual parameters (the nodes attached to the outer interface), the correspondence between actual and formal parameters being implicit in the sorting of outer and inner interfaces. Inner interfaces can also be handy for encoding polyadic input prefixes of process calculi, where the input variables are just local place-holders for the values to be received dynamically upon communication.

At the level of **AGN** syntax, this generalisation would correspond to introduce an *inner rank* for each $b \in B$ and to introduce terms like $b[\overline{z}.\mathbb{G}](\overline{y})$, where $\overline{z}$ are the nodes provided by the inner interface of b, whose sorting must match the inner rank of b. At the level of axiomatisation, the nodes provided by the inner interface should be α-convertible ($\overline{z}$ acts as a binder in $b[\overline{z}.\mathbb{G}](\overline{y})$, with scope $\mathbb{G}$) and correspondingly extended versions of axioms A8 and A10 should be given with suitable side-conditions (the notion of free nodes should also be updated).

At the level of the encoding in gs-monoidal theories, this would correspond to move from signatures to *hyper-signatures*, where operators $f \in \Sigma_{u,w}$ with $u, w \in S^*$ are allowed. Interestingly enough, gs-monoidal theories are quite stable and such an extension is seamless, as no additional axiom is required. This is not the case for term graphs, that are tailored to ordinary signatures. This could be annoying, because while we have seen that gs-monoidal theories over ordinary signatures play for term graphs the role played by Lawvere theories for ordinary terms (i.e., they neatly emphasise the essential algebraic structure underlying the set-theoretical presentation of term graphs), the gs-monoidal syntax itself can be hard to follow without the corresponding drawings, even for small terms (see for example Definition 16).

Nevertheless, in the case of hyper-signatures we can resort to consider an alternative model to term graphs, called *gs-graphs* [15], that is defined in terms of concrete (multi-)sets of *assignments*. More precisely, two kinds of assignment are allowed in gs-graphs: a *proper assignment* has the form $x'_1 : s'_1 \dots x'_k : s'_k := f(x_1 : s_1, \dots, x_h : s_h)$ (for $f \in \Sigma_{s_1 \cdots s_h, s'_1 \cdots s'_k}$), while an *auxiliary assignment* has either the form $x' : s := x : s$ (aliasing) or $!(x : s)$ (name disposal). Given a set of assignments A, when a name appears in the left member of an assignment we say that it is *assigned*, when it appears in the right member we say that it is *used* and write $x : s \sqsubset_A x' : s'$ if A contains an assignment where $x : s$ is used and $x' : s'$ is assigned (meaning that, to some extent, $x' : s'$ depends on $x : s$). Like term graphs, also gs-graphs come equipped with top and bottom interfaces: implicitly, the top interface is given by all names that are used but not assigned in A (called *leaves* in [15]), while the bottom interface contains all names $x' : s'$ assigned via an aliasing $x' : s := x : s$ in A (called *roots*). Each interface is ordered according to a fixed total order $\leq$ on sorted names.

A gs-graph A is *valid* if it satisfies all of the following: (1) every name is assigned at most once in A; (2) the transitive closure $\sqsubset_A^+$ of $\sqsubset_A$ is acyclic; (3) every $x' : s$ such that $x' : s := x : s$ belongs to A is a maximal element of $\sqsubset_A^+$;

(4) for each name n not assigned in A (exactly) one disposal $!(n)$ is present in A; (5) for each name n assigned in A no disposal $!(n)$ can be present in A. Then it can be shown that valid gs-graphs on the hyper-signature Σ (taken up to the intuitive notion of isomorphism induced by any injective name substitution that respects the total ordering $\leq$ on the names in the interfaces) are the arrows of the freely generated gs-monoidal category $\mathbf{GS}(\Sigma)$.

In summary, we are confident that the results of the previous sections can be generalised seamlessly by allowing edges with an inner rank in NR-graphs and in the terms of the algebra, and by exploiting gs-graphs rather than term graphs for the encoding.

7 Related Works

As recalled in the Introduction, graphs are widely used in Computer Science for a visual, intuitive representation of systems and models of any kind. Several notions of *hierarchical* graphs have been introduced along the years in various areas, often as a useful structuring mechanism to cope with the modelling of systems of realistic size. One of the earliest proposals are Harel's *higraphs* [18], used first for modelling database structures and next as a basis for statecharts. Several other such models have been proposed since then, for modelling database systems, object-oriented systems and hyper-media applications, among others (see, e.g., the recap in Section 7 of [9]).

In the realm of Graph Transformation Systems, the use of hierarchical graphs dates back to Pratt [21], who used them to represent data structures of programming languages. Several other models have been proposed since them, till the most recent and elaborated ones in [13,9]. The graphs of [13] share with our NR-graphs the fact that subgraphs are encapsulated in (hyper-)edges, but they do not allow arcs to cross edge boundaries. The approach of [9] is instead much more general than ours, because they provide separated representations of a system (given by a "flat" graph) and of its hierarchical structure (an acyclic graph), relating them with a "connection graph". Both these approaches will be sources of inspiration for the definition of graph transformation over our NR-graphs, but none of them provides an algebraic presentation.

More closely related to our proposal, and at the same time direct sources of inspiration for us, are some graph formalisms developed for modelling process calculi. They range from the several approaches based on flat graphs (see, e.g. [16]), with which we share the modelling of name restriction ν, to Milner's *bigraphs* [20]. Basically, a bigraph is given by the superposition of two graphs, representing the *locality* and the *connectivity structure* of a system, respectively, having the nodes in common. In our words, the first specifies the hierarchical structure of the system, while the second the naming topology. So, we do believe that the two approaches have essentially the same expressiveness, even if a precise comparison goes beyond the scope of this paper. It is worth noting, nevertheless, that the two approaches are in a sense dual to each other: bigraphs represent locations of a system as nodes (instead of hyper-edges) and names

as hyper-edges (instead of nodes): when designing our modelling framework we preferred to introduce the notion of NR-graph rather than to stick to bigraphs, because NR-graphs allow for a more intuitive representation of systems and have a much simpler definition w.r.t. bigraphs. During the revision of the present paper, we learned that an algebra for bigraphs has been proposed in [17]: we intend to study the precise relationship between this algebra and ours, to understand if the greater complexity of the former is balanced by a greater expressive power. Moreover, the algebra given in [17] is "fine-grained" and closer to the gs-monoidal algebra than to the **AGN** algebra, hence we think the result in this paper is an important step for relating NR-graphs and bigraphs.

Concerning the axiomatisation, several (sound and complete) axiomatisations of various families of graphs exist, and each of them provides a suitable linear syntax for the corresponding graphs. Most of the axiomatisation explicitly address node sharing, possibly following the seminal work on flownomial algebras [12]. It is not possible to mention here all the contributions to this field, but it seems noteworthy that all these structures, including the one discussed in Section 3 and proposed in [10], can be seen as enrichments of symmetric monoidal categories, which thus reasonably provide the basis for the description of distributed environments in terms of wire-and-box diagrams: see the survey [22] and the meta-formalism in [4].

Finally, it is worth stressing that the **AGN** algebra and the corresponding NR-graphs are very close to the algebra introduced in [7], whose semantics is defined set-theoretically over a suitable domain of hierarchical graphs with interfaces. Besides presenting a few technical differences (the hyper-edges of the algebra of [7] also offer an inner interface, like the one discussed in Section 6.2; edges without a nested graph are treated differently; there is only one type of (localised) restriction, and the extrusion of restricted names is handled with optional axioms) a formal encoding of that algebra into term graphs is not available yet, but should not be difficult with the formal background presented here. Instead, that algebra has been used in [5,6,7] to encode several process calculi featuring sophisticated notions of nesting and of restriction (including the π-calculus [19], Sagas [8], and CaSPiS [3], among others).

8 Conclusion and Further Works

In this paper we presented a simple model of hierarchical graphs featuring nesting of subgraphs within hyper-edges and two kinds of restrictions of nodes, which are suited for representing in a direct way a wide class of systems and models. In order to provide a linear, term-like syntax for such graphs, an axiomatisation has been proposed, the main result being that such axiomatisation is sound and complete. The result was proved by encoding such nested structures (both the algebra and the graphs) into a simpler model (term graphs) where the nesting is represented explicitly with a tree of locations, and by exploiting an existing axiomatisation of term graphs as gs-monoidal theories. Finally possible extensions of the presented framework are sketched, including the definition of a notion of

rewriting over nested graphs, and the generalisation of the graphical model to allow for edges with inner interfaces.

As topics of future research, besides those just mentioned we intend to clarify the formal relationship between our NR-graphs and Milner's bigraphs [20], and of our algebra with the one recently proposed in [17]. In parallel to this, we intend to test the adequacy of our modelling framework by encoding suitable algebraic formalisms (typically process calculi), which would automatically obtain a graphical representation, as well as visual modelling formalisms, for which we could obtain a handy linear syntax.

References

1. Baldan, P., Corradini, A., König, B.: A framework for the verification of infinite-state graph transformation systems. Information and Computation 206(7), 869–907 (2008)
2. Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M., Sleep, M.: Term graph reduction. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) PARLE 1987. LNCS, vol. 259, pp. 141–158. Springer, Heidelberg (1987)
3. Boreale, M., Bruni, R., Nicola, R.D., Loreti, M.: Sessions and pipelines for structured service programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
4. Bruni, R., Gadducci, F., Montanari, U.: Normal forms for algebras of connections. Theoretical Computer Science 286, 247–292 (2002)
5. Bruni, R., Corradini, A., Montanari, U.: Modeling a service and session calculus with hierarchical graph transformation (2010) (submitted)
6. Bruni, R., Gadducci, F., Lluch Lafuente, A.: An algebra of hierarchical graphs and its application to structural encoding. Scientific Annals in Computer Science (to appear, 2010)
7. Bruni, R., Gadducci, F., Lluch Lafuente, A.: A graph syntax for processes and services. In: Laneve, C., Su, J. (eds.) Web Services and Formal Methods. LNCS, vol. 6194, pp. 46–60. Springer, Heidelberg (2010)
8. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Palsberg, J., Abadi, M. (eds.) POPL 2005, pp. 209–220. ACM, New York (2005)
9. Busatto, G., Kreowski, H.J., Kuske, S.: Abstract hierarchical graph transformation. Mathematical Structures in Computer Science 15(4), 773–819 (2005)
10. Corradini, A., Gadducci, F.: An algebraic presentation of term graphs, via gs-monoidal categories. Applied Categorical Structures 7(4), 299–331 (1999)
11. Corradini, A., Montanari, U., Rossi, F.: An abstract machine for concurrent modular systems: CHARM. Theoretical Computer Science 122(1&2), 165–200 (1994)
12. Căzănescu, V.E., Ştefănescu, G.: A general result on abstract flowchart schemes with applications to the study of accessibility, reduction and minimization. Theoretical Computer Science 99(1), 1–63 (1992)
13. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. Journal on Computer and System Sciences 64(2), 249–283 (2002)
14. Ferrari, G.L., Hirsch, D., Lanese, I., Montanari, U., Tuosto, E.: Synchronised hyperedge replacement as a model for service oriented computing. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 22–43. Springer, Heidelberg (2006)

15. Ferrari, G.L., Montanari, U.: Tile formats for located and mobile systems. Information and Computation 156(1-2), 173–235 (2000)
16. Gadducci, F.: Graph rewriting for the pi-calculus. Mathematical Structures in Computer Science 17(3), 407–437 (2007)
17. Grohmann, D., Miculan, M.: Graph algebras for bigraphs. In: Ermel, C., de Lara, J., Heckel, R. (eds.) GT-VMT 2010. Electronic Communications of the EASST, vol. 29 (2010)
18. Harel, D.: On visual formalisms. Communication of the ACM 31(5), 514–530 (1988)
19. Milner, R.: Communicating and Mobile Systems. Cambridge University Press, Cambridge (1992)
20. Milner, R.: Pure bigraphs: Structure and dynamics. Information and Computation 204(1), 60–122 (2006)
21. Pratt, T.W.: Definition of programming language semantics using grammars for hierarchical graphs. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 389–400. Springer, Heidelberg (1979)
22. Selinger, P.: A survey of graphical languages for monoidal categories. In: Coecke, B. (ed.) New Structures for Physics. Lecture Notes in Physics. Springer, Heidelberg (to appear, 2010)

Stochastic Modelling and Simulation of Mobile Systems

Reiko Heckel and Paolo Torrini

Department of Computer Science, University of Leicester
{reiko,pt95}@le.ac.uk

Abstract. Distributed systems with mobile components are naturally modelled by graph transformations. To formalise and predict properties such as performance or reliability of such systems, stochastic methods are required. Stochastic graph transformations allow the integrated modelling of these two concerns by associating with each rule and match a probability distribution governing the delay of its application. Depending on the nature of these distributions, different techniques for analysis are available, ranging from stochastic model checking for exponential distributions to simulation for systems with general distributions.

In this paper we explore further the second notion, adding a model of global time to avoid the overhead of frequent local clock updates. We also transfer the notion of stochastic graph transformation system from an algebraic to a general background, allowing an arbitrary graph transformation approach to be used for the underlying system. We present a correspondingly extended semantic model, simulation algorithm and tool. The concepts are justified and illustrated by an accident management scenario which requires a model of physical mobility and sophisticated transformation concepts.

Keywords: mobile systems, stochastic simulation, graph transformation.

1 Introduction

To understand or design complex systems we are used to building models. Abstracting from irrelevant details of the subject systems, models render it intellectually and technically feasible to analyse them. However, since in this process information is lost, individual occurrences of events may become unpredictable, resulting in models that are nondeterministic both in the selection of actions and their timing. Stochastic methods allow us to make up for this lack of detail, analysing behaviour in terms of statistical aggregations instead of individual occurrences.

Graphs are among the simplest and most universal models for a variety of systems, not just in computer science, but throughout engineering and life sciences. When systems evolve, we are generally interested in the way they change, to predict, support, or react to evolution. Graph transformations combine the idea of graphs as a universal modelling paradigm with a rule-based approach

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 87–101, 2010.

to specify the evolution of systems. Supported by an established mathematical theory and a variety of tools for execution and analysis, graph transformation systems have been an object of study for more than 30 years.

As one of the early fathers and main promoters of this theory, Manfred Nagl started his studies on the expressive power of graph grammars and corresponding classes of formal languages of graphs in the early seventies [1]. In the late eighties and early nineties, Manfred Nagl pioneered the application of graph transformation in software engineering, specifically for the generation of software engineering tools [2]. The graph transformation language PROGRES [3] developed in this context provided one of the first scalable implementations of graph transformation and led to a wealth applications. It also provided a testbed for some of the advanced transformation concepts used in this paper, such as multi-objects and derived attributes. It turns out that these concepts are crucial for raising the level of abstraction of our models, and thus improving readability of specifications and scalability of analysis techniques. It is therefore only fitting to dedicate this paper to Manfred Nagl in appreciation of his contributions, on which we still rely today.

As motivated above, this paper uses graph transformation to model systems by a combination of structural changes and with stochastic time. This combination becomes more significant with the event of distributed and mobile systems, where volatile bandwidth and fragile connectivity as well as physical mobility make non-functional properties such as performance and reliability more and more important. To formalise, measure, and predict these properties, stochastic graph transformation systems have been introduced in [4].

They associate with each rule (and match) a probability distribution governing the delay of its application. Depending on the nature of these distributions, different techniques for analysis are available. In the case of exponential distributions, continuous-time Markov chains can be derived, which allow to verify stochastic properties through model checking.

In [5] we have illustrated this approach by an example of a P2P network. The property of interest here was the probability of finding the network in a condition where each participant could communicate with every other one, i.e., of the network graph being connected. A simple graph transformation model of network reconfigurations was used to generate the state space, bounded by an upper limit to the number of peers in the network. The resulting transition system was transformed into a Markov chain by associating rates with rule names.

This approach, while in line with existing practice, has a number of weaknesses.

- Model checking with explicit states is not suitable for models with large or infinite state space. Since performance and reliability may well depend on the overall scalability of the system, this is a serious limitation for verifying non-functional properties.
- Exponential distributions do not always provide the best abstraction for the timing of actions. For example, the time it takes to make a phone call or perform a communication handshake is more likely to follow a normal distribution.

– There are situations where the distributions would not only depend on the rules, but also on the graphs and matches they are applied to. For example, the time it takes to deliver a message may depend on the distance it has to travel, which may be an attribute of the connection.

Generalised stochastic graph transformation systems [6,7] allow for general distributions dependent on rules - match pairs (events, rather than just rules). Generalised semi-Markov processes provide a semantic model for such systems. Monte Carlo-style simulation allows for analysis, based on the execution of system runs probabilistically determined through the generation of pseudo-random numbers, rather than the full exploration of the state space. The results of stochastic analysis are statistical in nature, with confidence levels depending on the number of simulation runs. Thus, with respect to the validation of soft performance targets in large-scale systems, simulation based on semi-Markov processes may provide a more flexible tradeoff than model checking between analysis effort and confidence in the result.

However, in order for a component to measure, e.g., response time, local clocks had to be introduced. A local clock is represented by an attribute *chronos* for storing the current value and a *clock tick* rule such as the one in Fog. 1 to advance the attribute by one unit of time. The rule is to be used with a normal distribution, with mean given by one unit of time and variance the average deviation of the clock in one step. This approach, while sound, is unsuitable for simulating larger systems. Imagine, for example, a system of 100 components, each equipped with a local clock, and assume milliseconds as the relevant unit of time. In this case it would take 100,000 applications of rule clock tick to simulate the passing of one second of real time.

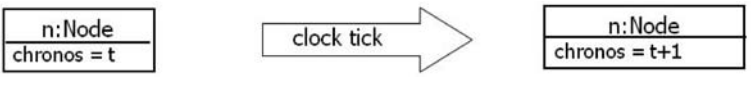


Fig. 1. Rule for advancing the local clock of a component

In this paper we will avoid this overhead by using a global notion of time. More precisely, whenever a rule is applied it will have access to the present simulation time to compute values assigned to attributes in the right-hand side of the rule. The need for such an extension, as well as for a number of other advanced features, is illustrated by the scenario of an accident management system deploying emergency vehicles to serve accidents. We also discuss how this extension of the semantic model impacts on the simulation algorithm and present a simulation tool developed in Java-Eclipse as an extension of the graph transformation tool VIATRA [8].

The paper is organised as follows. Section 2 below discusses related work on stochastic simulation. Then, Section 3 presents our application scenario and derives requirements for both the underlying graph transformation approach and

its stochastic extension. Section 4 introduces formally stochastic graph transformation system before Section 5 describes their semantics and simulation and Section 6 concludes the paper.

2 Related Work

The definition of generalised stochastic graph transformation systems that we are presenting here, with their semi-Markov process semantics based on a translation to generalised semi-Markov schemes, generalises previous proposals presented in [6,7]. Stochastic graph transformation systems (SGTS) as introduced in [9] allow only for exponential distributions associated with rule names, and have a semantics based on Markov chains. We depart from [6] by making the probability distribution dependent on the event (rule name and match) rather than just on the rule name — therefore drawing a closer parallel with the representation of semi-Markov processes as generalised semi-Markov schemes we rely on for the semantic interpretation [10].

At a technical level, our approach relies on a choice of a deterministic rule application operator, giving it a distinctly implementation-oriented flavour. On the other hand, in [6] the set of events is provided abstractly by a global name space given by the unfolding of the graph transformation system. Unlike [7], we introduce a notion of global time, leading to synchronous features comparable to null-delay transitions in generalised stochastic Petri nets [11].

Rule-based modelling based on graph transformation has been proposed as basis for stochastic simulation of biochemical systems [12]. Stochastic simulation based on the Gillespie algorithm [13] uses exponential distributions and is closely associated to multiset rewriting. Related approaches to simulation of stochastic Petri nets allow for transitions associated with probabilistic values [11]. Semi-Markov Petri-nets [14] allow for non-exponential distributions, too. Graph rewriting is more general than multiset rewriting, i.e., graphs can be seen as multisets of nodes *with edges between them*. However, [15] gives a method to map graph transformations into Petri-net transitions for a limited class of systems.

A probabilistic rewriting logic with rule matches associated to generic probabilistic values has been implemented in Maude [16], allowing for the simulation. While rewriting logic can encode graph transformation, the model does not describe time but probabilities of different outcomes of decisions. High-level implementations of stochastic simulation based on process algebra have been given, with PEPA [17] and stochastic π-calculus [18], allowing for exponential rates to be associated to actions. Semi-Markov PEPA [19] and the extended stochastic π-calculus presented in [20] allow for non-exponential distributions. While PEPA models are limited to finite-state systems, π-calculus has a similar level of expressive power like graph transformation. The main difference is in the style of specification, i.e., graph transformation as a high-level visual language vs. π-calculus as a programming-oriented process calculus.

3 Case Study and Requirements

In order to justify and illustrate our requirements we are considering a case study of an accident management system loosely based on a scenario developed in the IST FP6 project SENSORIA on *Software Engineering for Service-Oriented Overlay Computers*. The objective of the case study is to study the interplay of dynamic reconfiguration in the system with quality attributes and physical mobility.

We assume that vehicles may be subscribed to a service that, in the case of an accident (e.g., when the airbag is triggered) alerts the emergency services. They will check the alert by contacting the driver of the car on their mobile phone and, in case of need, dispatch an ambulance and/or recovery services. To minimise the impact of the accident, the car should be removed as quickly as possible in order to prevent other cars travelling behind from being stopped or slowed down.

The model has to account for a divers range of activities such as the physical movement of vehicles on the road network, the occurrence of accidents and sending of alerts to emergency services, and the assignment of emergency vehicles. A typical question to be asked is, what is the best strategy for serving accidents: Should they be served in chronological order, based on their distance from the nearest available emergency vehicle, or based on their perceived urgency, e.g., the number of cars caught up in tailback, etc? Another question is if, based on certain assumptions in the model, service-level guarantees can be given, such as "95% of accidents will have an emergency vehicle arriving at the scene within 20min".

We will describe and model the case study in more detail below. However, this informal description is already sufficient to identify a number of characteristic features of our modelling approach. We will use *graphs to represent system states*, i.e., the road network, vehicles and their current, as well as logical relationships between them, such as a car having suffered an accident on a road, or an emergency vehicle being assigned to serve an accident. To record the exact position of a car or the number of cars caught up in a tailback we will require *attributed graphs*. While the road network should be static, vehicles and their relationships are subject to change. This change is best modelled by *graph transformation*.

We will further require a model of time. However since, e.g., the time between two occurrences of accidents in a certain area is not deterministic, a stochastic model of time is required. Moreover, different types of probability distributions will be required. For example, the time between two (consecutive but independent) accidents is governed by an exponential distribution whose rate is reciprocal of the average time between two accidents, while the time it takes an emergency vehicle to arrive at the scene of an accident should have a normal distribution with the average delay as mean. In some cases, the parameters of these distributions will depend on the graphs and matches, i.e., the arrival time could depend on the lengths of the shortest route to be travelled. Moreover, attribute assignments and conditions will need access to the current simulation time, e.g., to record the time when an accident happened.

In order to determine shortest routes in the road network, e.g., to guide vehicles or find the nearest one to an accident, complex computations will have to be carried out on the graph representing the state. Modelling these at the same level as, e.g., the movement of cars or the sending of alerts, we would run the risk of obfuscating the model with details that are better addressed by employing standard algorithmic solutions. We will have to provide an interface to "invoke" such solutions without cluttering our rules. In general, since we are interested in a model that can be analysed for the kinds of quality-of-service properties described above, so we should avoid increasing the state space of our model by complex computations.

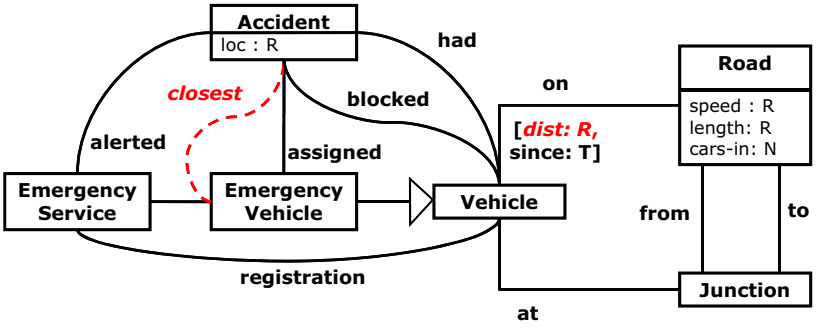


Fig. 2. Attributed type graph with inheritance, derived attributes and edges

The type graph of our model is shown in Fig 2. Beside node and edge types, it contains node attribute declarations such as the average *speed* of a road, its *length*, or *cars-in*, the number of cars that have entered it. Edges can be attributed as well, e.g., the edge representing the location of a car *on* a road records the point in time *since* the car has entered it.

The model uses inheritance of node types to distinguish special emergency vehicles that are assigned to accidents by emergency services. In general, vehicles can be involved in accidents or blocked by them.

Finally, we make use of *derived* attributes such as the *dist*ance of the car from the start of the road it is travelling on, and derived associations such as the *closest* available emergency vehicle for an accident. The former is computed by evaluating the following definition whenever the attribute is accessed

$$on.dist = on.Road.speed * (\text{now} - on.since)$$

with the constant *now* referring to the current simulation time. The derived link, which points to the emergency vehicle with the quickest route to the accident, is calculated *on demand* based on a standard shortest path computation.

Such derived attributes are well known in PROGRES [3]. They can be easily encoded, albeit with a loss of elegance, in graph transformation languages providing control flow constructs that allow invocation of rules from other rules.

We prefer the abstract and declarative notation of derived attributes and links at the model level, but will have to rely on a corresponding encoding when implementing the rule in VIATRA.

4 Generalised Stochastic Graph Transformation

In this section, we provide the basic notions of stochastic graph transformation together with their semantic model of generalised semi-Markov processes. While we present general definitions with respect to a generic attributed graph transformation approach, examples will be based on typed attributed graph transformation with SPO-like semantics, allowing for node type inheritance and negative application.

4.1 The Base: Graph Transformation, Generically

In existing axiomatic descriptions of graph transformation [21], a graph transformation approach is given by a class of graphs $\mathcal{G}$, a class of rules $\mathcal{R}$, and a family of binary relations $\Rightarrow_r : \mathcal{G} \times \mathcal{G}$ representing transformations by rules $r \in \mathcal{R}$.

We refine this presentation as follows. First, we assume an $\mathcal{R} \times \mathcal{G}$-indexed family of sets of rule matches $\mathcal{M}_{r,G}$ and extend $\Rightarrow_r$ to a partial function $\Rightarrow_{r,m}$: $\mathcal{G} \to \mathcal{G}$, such that $\Rightarrow_{r,m} (G)$ is defined if and only if $m \in \mathcal{M}_{r,G}$. This captures the idea that rule application is well-defined and deterministic once a valid match m for r in G is found. In case r is equipped with application conditions, the match is deemed to satisfy them. As usual, we write $G \Rightarrow_{r,m} H$ for $\Rightarrow_{r,m} (G) = H$.

We denote by $\mathcal{M}_r$ the set of *all* matches for a rule r, and by $\mathcal{M}_R = \bigcup_{r \in R} \mathcal{M}_r$ the union of all such sets of matches for all rules in $R \subseteq \mathcal{R}$. Similarly, $\mathcal{M}_{R,G}$ is the subset of $\mathcal{M}_R$ of matches into a graph $G \in \mathcal{G}$ only.

In general however, and in contrast to the strongly typed notion of match in the categorical presentation of matches as morphisms favoured in the algebraic approaches [22,23], we assume that a match depends on the rule, but not strictly on the target graph. Indeed, a match for a rule r may only use a part of a graph G. In this case, the same match can also be defined into a graph G' if G' shares the part of G used by m. This is important, for example, if a match for a rule r has already been established, but an application of another rule r' is scheduled to take place first. If the applications of r and r' are independent, the match for r will still be valid after the application of r', even if the graph has changed outside the scope of this match.

Such a general concept of match is required to define a notion of event as a rule-match pair transcending individual states. In [6,7] such a goal was achieved by introducing an equivalence relation on matches. Two matches m and m' for the same rule r are equivalent if they define the same occurrence of r's left-hand side in graphs G, G' related by a transformation $G \Rightarrow G'$ preserving the occurrence. In this way, events can be associated to equivalence classes of rule matches, and thus outlive the target graph of the match if the change does not affect the elements in the codomain.

Moreover, for algebraic approaches based on categorical constructions, where rule application is only defined up to isomorphism, further assumptions have to be made in order to make rule application deterministic: To capture the idea of a concrete and deterministic implementation of graph transformation, a choice of direct transformations is made such that the result of applying a rule to a match is unique, and names are chosen consistently in consecutive transformations preserving the identities of nodes and edges where possible and never reusing names from the past.

In its simplest form, a graph transformation system $\mathbf{G} = \langle R, G_0 \rangle$ consists of a set $R \subseteq \mathcal{R}$ of rules and an initial graph $G_0 \in \mathcal{G}$. A transformation in $\mathbf{G}$ is a sequence of rule applications $G_0 \Rightarrow_{r_1,m_1} G_1 \Rightarrow_{r_2,m_2} \cdots \Rightarrow_{r_n,m_n} G_n$ using rules in $\mathbf{G}$ with all graphs $G_i \in \mathcal{G}$. The set of graphs reachable from G_0 by rules in R is denoted by $\mathcal{L}(\langle R, G_0 \rangle)$.

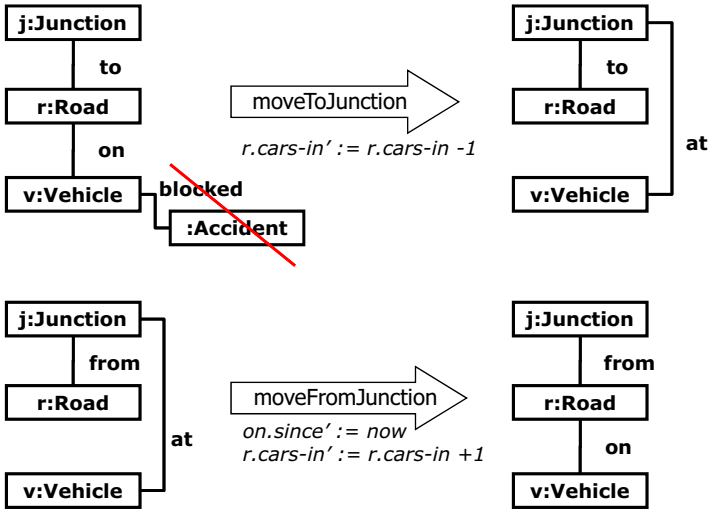


Fig. 3. Rules for moving

4.2 The Topping: Stochastic Modelling with Global Time

We can intuitively rely on standard notions of stochastic process and discrete event system [10], and treat a graph transformation system as a discrete event systems where events are rule matches. The stochastic aspect can be introduced by associating each event with a distribution function governing the delay in the application of the corresponding transformation step.

Transformations will be aware of the passage of time, so that attribute values can be computed using the current time as parameter. For this purpose, we define a notion of timed execution of a graph transformation system. Given a

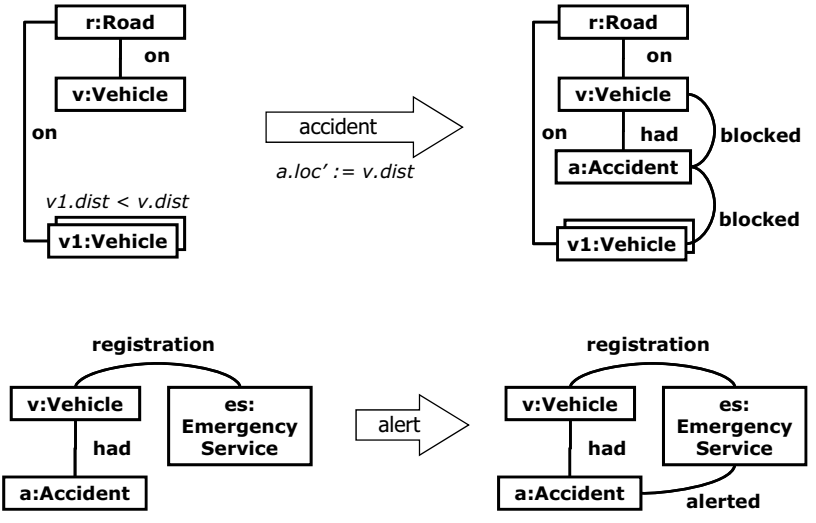


Fig. 4. Accidents happen and get reported

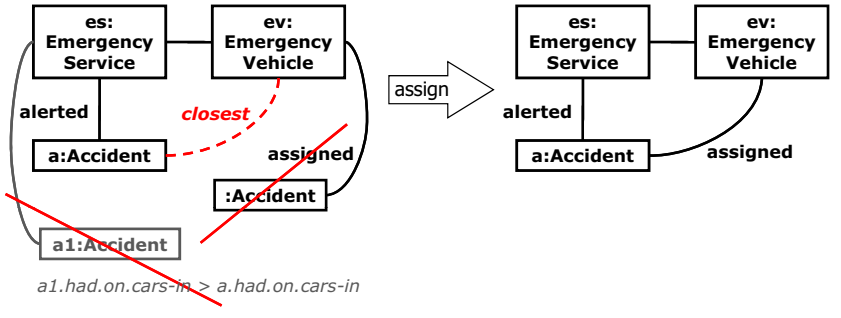


Fig. 5. Assigning the closest emergency vehicle

transformation sequence $s = G_0 \Rightarrow_{r_1,m_1} G_1 \Rightarrow_{r_2,m_2} \cdots \Rightarrow_{r_n,m_n} G_n$, a timed execution of s is a sequence

$$G_0 \Rightarrow_{r_1,m_1,t_1} G_1 \Rightarrow_{r_2,m_2,t_2} \cdots \Rightarrow_{r_n,m_n,t_n} G_n$$

such that $t_i < t_j$ for $1 \leq i < j \leq n$. In practice, the times t_i for will be provided by the simulation environment in such a way that, statistically, they are consistent with the distribution of delays specified by the stochastic graph transformation systems.

For a set of rules R, by $\mathcal{E}_R = \{\langle r, m\rangle \mid m \in \mathcal{M}_r, r \in R\}$ we denote the set of events, given by pairs of rules and matches. Then $\mathbf{S} = \langle \mathbf{G}, F\rangle$ is a *stochastic graph transformation system* (SGTS) where $\mathbf{G}$ is a graph transformation system and $F : \mathcal{E}_R \to \mathbf{R} \to [0, 1]$ is a function which associates with every match for a

rule in **G** a probability distribution function. We assume $F(e)(0) = 0$ (*null delay* condition).

The behaviour of a stochastic graph transformation system can be described as a stochastic process over continuous time, where reachable graphs form a discrete state space, the application of transformation rules defines state transitions as instantaneous events, and interevent times, determined by the application of transformation rules, are dominated by continuous probability distributions. More precisely, we associate each rule, as it becomes enabled by a match, with an independent random variable (*timer*), which represents the time expected to elapse (*scheduled delay*) before the rule match is applied, and is associated with a cumulative distribution function (*cdf*). At runtime, the timer is randomly assigned a value according to its *cdf*.

Note that, in order to avoid cyclic dependencies, distribution functions and rule matches are independent of the current simulation time, i.e., the value of *now* is only used to determine attribute values in the new state. This happens, for example, in rule *moveFromJunction* with the assignment $on.since := now$ and indirectly in rule *accident* with $a.loc := v.dist$ due to the computation of time-dependent derived attribute $v.dist$.

Let us consider the distributions to be assigned to the rules in our case study.

- Rule *moveToJunction* should be governed by a normal distribution with $mean = r.length/r.speed$, the expected time taken to travel the length of the road. The variance should reflect the expected average deviation. Note that this is an example where the distribution's parameters depend on the match.
- Rule *moveFromJunction* should be controlled by a normal distribution, with fixed mean and deviation.
- Rule *accident* is a typical example of an event that is exponentially distributed, its rate reciprocal to the average time between two accidents.
- Rules *alert* and *assign* represent computation and communication actions, which should be distributed normally with fixed parameters.

Such definitions provide the data for function $F : \mathcal{E}_R \to \mathbf{R} \to [0, 1]$ in the notion of stochastic graph transformation system above.

5 Stochastic Simulation

Next, we introduce the semantic model as well as operational interpretation of stochastic graph transformation systems in terms of suitable generalisations of semi-Markov processes and stochastic simulation.

5.1 From SGTS to Generalised Semi-Markov Processes

We rely on discrete event semantics of stochastic processes, which essentially consists of finite state systems with probabilistic transitions associated with timed events [10]. In generalised semi-Markov processes (GSMP), timers as well

as the resulting interevent times can be generally distributed. This corresponds to a model in which events are generally independent of the past states but, unlike Markov processes, they may depend on interevent times. More formally, a GSMP can be defined as a process generated by a *generalised semi-Markov scheme* (GSMS) [24]. Here we define a notion which actually extends that of GSMS. A *timed GSMS* (TGSMS) is a structure

$$\mathcal{P} = \langle\ Z,\ E,\ active : Z \to \wp E,\ new : Z \times E \to \mathbf{R} \to Z,$$
$$\Delta : E \to \mathbf{R} \to [1, 0],\ s_0 : Z\ \rangle$$

where Z is a set of system states; E is a set of implicitly timed events; *active* is the activation function, so that $active(s)$ is the finite set of active events associated with s; *new* is the transition function depending on states, events and time values; Δ is the distribution assignment, so that $\Delta(e)$ is the general cumulative distribution function associated with the scheduled delay of event e; and s_0 is the initial state.

Unlike the notion defined in [24], where $new : Z \times E \to Z$, we assume that that *new* is a timed function — taking also the scheduled time of the state transition (corresponding to the value of *now*) as a parameter. This might be regarded as an extension to cover the case in which there are functions $\mathbf{R} \to Z$ instead of states. Notice, however, that *active* ignores this dependency, i.e., the activation of events is not affected by time scheduling — this goes together with the fact that in the graph transformation systems, *now* is used to determine the value of attributes in the new state, but never in application conditions that determine activation.

GSMPs are generalisations of continuous-time Markov chains — they can be regarded as Markov processes with generally distributed timers [25]. Indeed, a process generated by a GSMS where all timers are exponentially distributed variables is stochastically equivalent to a continuous-time Markov chain — this follows from the memoryless property enjoyed by exponential distributions

$$P(X > x + z | X > z) = P(X > x)$$

and therefore from the fact that, from the point of view of stochastic analysis, exponentially distributed timers can be restarted at each step — which means that no matter how long an event has been scheduled for, until it does not happen, the probability of experiencing a further delay remains the same.

Given a stochastic graph transformation system $\mathbf{S} = \langle R, G_0, F \rangle$, we can define its translation as a TGSMS

$$\Pi(\mathbf{S}) = \langle S, E, active, new, \Delta, G_0 \rangle$$

where

- $S = \mathcal{L}(\langle R, G_0 \rangle)$ is the set of graphs reachable from G_0 via rules in R;
- $E = \mathcal{E}_R$ is the set of possible events for R;
- $active(G) = \{\langle r, m \rangle \mid m \in \mathcal{M}_{r,G}\} \subseteq E$ is the set of all events enabled in graph G;

- the transition function is defined by $new(G, \langle r, m\rangle)(t) = H$ if $G \Rightarrow_{r,m,t} H$;
- distributions $\Delta(\langle r, m\rangle)$ are given by $F(\langle r, m\rangle)$;
- G_0 is the initial graph in $\mathbf{S}$.

The embedding of SGTS into TGSMS can be used as theoretical framework for the definition of a simulation algorithm that is adequate with respect to system runs. In fact, given an SGTS $\mathbf{S}$ and its translation $\mathcal{P} = \Pi(\mathbf{S})$, it is not difficult to see, reasoning by induction on the number of steps, that for each transformation from G_0 in $\mathbf{S}$, this can be simulated by a sequence of transitions in $\mathcal{P}$ — and vice-versa.

5.2 Algorithm and Tool for Stochastic Simulation

Given a TGSMS $\mathcal{P} = \langle Z,\ E,\ active,\ new,\ \Delta, s_0\rangle$ with global time, an algorithm for its simulation can be described as follows, based on the general scheduling scheme given in [10].

- For the initial step
  1. The simulation time is initialised to 0.
  2. The set of the activated events $A = active(s_0)$ is computed.
  3. For each event $e \in A$, a scheduling time t_e is computed by adding 0 to a random delay value d_e given by the random number generator (RNG) depending on the probability distribution function $\Delta(e)(0)$;
  4. The active events with their scheduling times are collected in the scheduled event list $l_{s_0} = \{(e, t_e) | e \in A\}$, ordered by the time values.
- For every other step — given the current state $s \in S$ and the associated scheduled event list $l_s = \{(e, t) | e \in active(s)\}$
  1. the first element $k = (e, t)$ is removed from l_s;
  2. the simulation time t_S is updated by increasing it to t;
  3. the new state s' is computed as $s' = new(s, e)(now)$;
  4. the list $m_{s'}$ of the surviving events is computed, by removing from l_s all the elements become inactive, i.e. all the elements (z, x) of l_s such that $z \notin active(s')$;
  5. a list $n_{s'}$ of the newly activated events is built, containing a single element (z, t) for each event z such that $z \in active(s') \backslash active(s)$ and has scheduling time $t = t_S + d_z$, where d_z is a random delay value given by the RNG depending on the distribution function $\Delta(z)(t_S)$;
  6. the new scheduled event list $l_{s'}$ is obtained by reordering the concatenation of $m_{s'}$ and $n_{s'}$ with respect to the time values

Based on this scheduling scheme and the embedding of SGTS into GSMS with global time, a simulation tool has been developed supporting the analysis of stochastic graph transformations. The implementation of stochastic simulation based on graph transformation poses computational challenges — the biggest one being the computation of the set of all matches at each step. We rely on the incremental pattern-matching [26] as implemented in VIATRA to cope with this problem.

VIATRA [8] is a graph transformation tool available as *Eclipse* plug-in, which supports attributes, negative conditions, and all the other advanced transformation concepts required. VIATRA's RETE-style incremental pattern-matching approach consists of storing pre-computed pattern matching information in a tree structure that gets updated at each transformation step, rather than computing matches from scratches at each step. This approach allows to compute matches in constant time with respect to the graph size, for the penalty of increased memory consumption.

The simulation tool is based on the integration between this graph transformation engine and an implementation of the general scheduling scheme. For the latter we rely on the SSJ stochastic libraries [27] for both random number generation and statistic functions.

Like VIATRA itself, the simulation tool is implemented as *Eclipse* plugin. While the underlying graph transformation system is encoded as a textual VIATRA model transformation, probability distributions and their parameters are specified as model elements, such as the number of runs, their limit in either time or number of steps, the name of the rule set with associated stochastic parameters, and a distinguished value to allow running a suite of simulations with varying parameters. Experiments based on a logically simpler but computationally as demanding model are encouraging as far as performance is concerned.

6 Conclusion

This paper introduces generalised stochastic graph transformation with global time as a technique to model stochastic processes with dynamic reconfiguration. These systems can be translated into generalised semi-Markov schemes with time, an embedding which is useful in two respects. From the theoretical point of view it gives a semantical framework for SGTS. From the application point of view it provides a general algorithm to implement simulation.

Instantiating and specialising this algorithm, a simulation environment has been developed based on the VIATRA graph transformation tool which provides an efficient implementation of an expressive language suitable to create models at a high level of abstraction. We are presently in the process of evaluating the performance of this tool on large and more complex case studies.

Acknowledgments. This work has been partially supported by the project SENSORIA, IST-2005-016004. We are indebted to István Ráth for his help with installing, using, and extending the VIATRA tool environment.

References

1. D'Argenio, P.R., Katoen, J.P.: Formal languages of labelled graphs. Computing 16, 113–137 (1976)
2. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996)

3. Schürr, A.: Introduction to PROGRES, an attribute graph grammar based specification language. In: Nagl, M. (ed.) WG 1989. LNCS, vol. 411, pp. 151–165. Springer, Heidelberg (1990)
4. Heckel, R., Lajios, G., Menge, S.: Stochastic graph transformation systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 210–225. Springer, Heidelberg (2004)
5. Heckel, R.: Stochastic analysis of graph transformation systems: A case study in P2P networks. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 53–69. Springer, Heidelberg (2005)
6. Kosiuczenko, P., Lajios, G.: Simulation of generalised semi-Markov processes based on graph transformation systems. Electr. Notes Theor. Comput. Sci. 175(4), 73–86 (2007)
7. Khan, A., Torrini, P., Heckel, R.: Model-based simulation of VoIP network reconfigurations using graph transformation systems. ECEASST 16, 1–20 (2008)
8. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA model transformation system. In: GRaMoT 2008: Proceedings of the Third International Workshop on Graph and Model Transformations, pp. 25–32. ACM, New York (2008)
9. Heckel, R., Lajios, G., Menge, S.: Stochastic graph transformation systems. Fundamenta Informaticae 74 (2006)
10. Cassandras, C.G., Lafortune, S.: Introduction to discrete event systems. Kluwer, Dordrecht (2008)
11. Marsan, M.A., Bobbio, A., Donatelli, S.: Petri nets in performance analysis: An introduction. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 211–256. Springer, Heidelberg (1998)
12. Danos, V., Feret, J., Fontana, W., Krivine, J.: Scalable simulation of cellular signaling networks. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 139–157. Springer, Heidelberg (2007)
13. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry, 2340–2361 (1977)
14. Bradley, J.T., Dingle, N., Harrison, P.G., Knottenbelt, W.J.: Performance queries on semi-Markov stochastic Petri nets with an extended continuous stochastic logic. In: 10th International Workshop on Petri Nets and Performance Models, pp. 1063–1067 (2003)
15. Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Rule-based modelling, symmetries, refinements. In: Fisher, J. (ed.) FMSB 2008. LNCS (LNBI), vol. 5054, pp. 103–122. Springer, Heidelberg (2008)
16. Kumar, N., Koushik, S., Meseguer, J., Gul, A.: A rewriting based model for probabilistic distributed object systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 32–46. Springer, Heidelberg (2003)
17. Gilmore, S., Hillston, J.: The PEPA workbench: a tool to support a process algebra-based approach to performance modelling. In: Haring, G., Kotsis, G. (eds.) TOOLS 1994. LNCS, vol. 794, pp. 353–368. Springer, Heidelberg (1994)
18. Priami, C.: Stochastic pi-calculus. The Computer Journal 38(7), 578–589 (1998)
19. Bradley, J.T.: Semi-Markov PEPA: modelling with generally distributed actions. International Simulation Journal 6(3-4), 43–51 (2005)
20. Priami, C.: Stochastic pi-calculus with general distributions. In: Proc. of the 4th Workshop on Process Algebras and Performance Modelling (PAPM 1996), CLUT, pp. 41–57 (1996)

21. Kreowski, H.J., Kuske, S.: On the interleaving semantics of transformation units - a step into GRACE. In: Cuny, J., Engels, G., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1994. LNCS, vol. 1073, pp. 89–106. Springer, Heidelberg (1996)
22. Ehrig, H., Pfender, M., Schneider, H.: Graph grammars: an algebraic approach. In: 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 167–180. IEEE, Los Alamitos (1973)
23. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer, Heidelberg (2006)
24. D'Argenio, P.R., Katoen, J.P.: A theory of stochastic systems part I: Stochastic automata. Inf. Comput. 203(1), 1–38 (2005)
25. Nelson, R.: Probability, Stochastic processes, and queueing theory. Springer, Heidelberg (1995)
26. Varró, G., Varró, D.: Graph transformation with incremental updates. Electr. Notes Theor. Comput. Sci. 109, 71–83 (2004)
27. L'Ecuyer, P.L., Meliani, L., Vaucher, J.: SSJ: a framework for stochastic simulation in Java. In: Proceedings of the 2002 Winter Simulation Conference, pp. 234–242 (2002)

Autonomous Units and Their Semantics – The Concurrent Case*

Hans-Jörg Kreowski and Sabine Kuske

University of Bremen, Department of Computer Science
P.O. Box 330440, D-28334 Bremen, Germany
{kreo,kuske}@informatik.uni-bremen.de

Abstract. Communities of autonomous units are rule-based and graph-transformational devices to model processes that act and interact, move and communicate, cooperate and compete in a common environment. The autonomous units are independent of each other, and the environment may be large and structured in such a way that a global synchronization of process activities is not reasonable or not feasible. To reflect this assumption properly, a concurrent-process semantics of autonomous units is introduced and studied in this paper employing the idea of true concurrency. In particular, causal dependency between actions of autonomous units is compared with shift equivalence known from graph transformation, and concurrent processes in the present approach are related to canonical derivations.

1 Introduction

In this paper, we introduce and investigate the concurrent semantics of autonomous units. Communities of autonomous units are proposed in [7] as rule-based and graph-transformational devices to model interactive processes that run independently of each other in a common environment. An autonomous unit has a goal that it tries to reach, a set of rules the applications of which provide its actions, and a control condition which regulates the choice of actions to be performed actually. Each autonomous unit decides about its activities on its own right depending on the state of the environment and the possibility of rule applications, but without direct influence of other ongoing processes.

In [9], the sequential as well as the parallel semantics of autonomous units is studied. In the sequential case, a single unit can act at a time while all other units must wait. This yields sequences of rule applications interleaving the activities of the various units. Typical examples of this kind are board games with several players who can perform their moves in turn. In the parallel case, the process steps are given by the application of parallel rules that are composed of the rules of the active units. In this way, units can act simultaneously providing a kind

* Research partially supported by the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 102–120, 2010.

of parallelism which is known from Petri nets, cellular automata, multi-agent systems, and graph transformation.

The sequential and the parallel semantics of communities of autonomous units are based on sequential and parallel derivations respectively. Both are composed of derivation steps. In other words, the semantics assumes implicitly the existence of a global clock to cut the run of the whole system into steps. But this is not always a realistic assumption, because the environment may be very large and – more important – the idea of autonomy conflicts with the regulation by a global clock. For example, trucks in a large transport network upload, move, and deliver asynchronously, and do not operate in simultaneous steps and even less in interleaved sequential steps.

The concurrent semantics avoids the assumption of a global clock. The actions of units are no longer totally ordered or simultaneous, but only partially ordered. The partial order reflects causal dependencies meaning that one action takes place before another action if the latter needs something that the former provides. The causal dependency relation of the concurrent semantics of autonomous units is compared with shift equivalence known from graph transformation, and concurrent processes in the present approach are related to canonical derivations (see also [11,3,2]).

The paper is organized as follows. Section 2 contains some preliminaries concerning multisets and graphs. In Section 3, concurrent graph transformation approaches are introduced which provide the basic ingredients of autonomous units. In Section 4, communites of autonomous units are presented and a concurrent semantics is defined for them. Section 5 is dedicated to canonical derivations which constitute a kind of representantives for the concurrent runs in a community. The introduced concepts are illustrated with a running example solving a generalization of the well-known Hamiltonian path problem. The last section contains the conclusion.

2 Preliminaries

In this section, we recall some basic notions and notations concerning multisets and graphs as far as they are needed in this paper.

Multisets. Given some basic domain D, the set of all multisets D_* over D with finite carriers consists of all mappings $m\colon D \to \mathbb{N}$ such that the carrier $car(m) = \{d \in D \mid m(d) \neq 0\}$ is finite. For $d \in D$, $m(d)$ is called the multiplicity of d in m. The union or sum of multisets can be defined by adding corresponding multiplicities, i.e., $m + m'(d) = m(d) + m'(d)$ for all $m, m' \in D_*$ and $d \in D$. D_* with this sum is the free commutative monoid over D where the multiset with empty carrier is the null element, i.e. $\mathbf{0}\colon D \to \mathbb{N}$ with $\mathbf{0}(d) = 0$ for all $d \in D$. Note that the elements of D correspond one-to-one to singleton multisets, i.e. for $d \in D, \hat{d}\colon D \to \mathbb{N}$ with $\hat{d}(d) = 1$ and $\hat{d}(d') = 0$ for $d' \neq d$. These singleton multisets are the generators of the free commutative monoid. This means in particular that, for every $m \in D_*$, there are $d_1, \ldots, d_k \in D$ with $m = \sum_{i=1}^{k} \widehat{d_i}$.

Graphs. A (directed edge-labeled) graph is a system $G = (V, E, s, t, l)$ where V is a set of nodes, E is a set of edges, $s, t: E \to V$ assign to every edge its source $s(e)$ and its target $t(e)$, and the mapping l assigns a label to every edge in E. The components of G are also denoted by V_G, E_G, etc. As usual, a graph M is a subgraph of G, denoted by $M \subseteq G$ if $V_M \subseteq V_G$, $E_M \subseteq E_G$, and s_M, t_M, and l_M are the restrictions of s_G, t_G, and l_G to E_M. A graph morphism $g: L \to G$ from a graph L to a graph G consists of two mappings $g_V: V_L \to V_G$, $g_E: E_L \to E_G$ such that sources, targets and labels are preserved, i.e. for all $e \in E_L$, $g_V(s_L(e)) = s_G(g_E(e))$, $g_V(t_L(e)) = t_G(g_E(e))$, and $l_L(e) = l_G(g_E(e))$. In the following we omit the subscript V or E of g if it can be derived from the context. In order to represent also graphs that contain labeled as well as unlabeled edges, we assume the existence of a special symbol $\diamond$. Every edge labeled with $\diamond$ is then regarded as an *unlabeled edge*. All other edges are *labeled edges*. An edge is called a *loop* if its source and target coincide. In graphical representations we omit the loops, i.e., their labels are placed next to their sources. If the labeled loops of a node is the set $\{e_1, \ldots, e_k\}$ where for $i = 1, \ldots, k$ the label of e_i is x_i ($x_i \neq \diamond$), the node will be called a $\{x_1, \ldots, x_k\}$-node. In the case where $k = 1$, the node is called an x_1-node. A node without any labeled loop is called a λ-node.

3 Concurrent Graph Transformation Approaches

Graph transformation (see, e.g., [18,6,1]) constitutes a formal specification framework that supports the modeling of the rule-based transformation of graph-like, diagrammatic, and visual structures. The ingredients of graph transformation are provided by so-called graph transformation approaches. In this section, we recall the notion of a graph transformation approach as introduced in [12], but modified with respect to the purposes of this paper.

Two basic components of every graph transformation approach are a class of graphs and a class of rules that can be applied to these graphs. In many cases, rule application is highly nondeterministic – a property that is not always desirable. Hence, graph transformation approaches can also provide a class of control conditions so that the degree of nondeterminism of rule application can be reduced. Moreover, graph class expressions can be used in order to specify sets of initial and terminal graphs of graph transformations. In the following, transformations from initial to terminal graphs via rule applications according to control conditions are called (graph transformation) processes.

The basic idea of parallelism in a rule-based framework is the application of many rules simultaneously and also the multiple application of a single rule. To achieve these possibilities, we assume that multisets of rules can be applied to graphs rather than single rules. If $\mathcal{R}$ is a set of rules, $r \in \mathcal{R}_*$ comprises a selection of rules each with some multiplicity. Therefore, an application of r to a graph yields a graph which models the parallel and multiple application of several rules.

If there is no global clock and no synchronization mechanism that cuts the actions (i.e., the rule applications) of processes into steps, then the process actions are not totally ordered, but only partially. An action occurs necessarily before another action if the former one generates something that the latter one needs. Moreover, an action that removes something that is needed by another action prohibits the latter one to occur if the former one is performed first. Both situations describe causal dependencies. In all other cases, it may be impossible to establish the order of time of two actions: They may occur one after the other in any order or even in parallel. This is the idea of true concurrency which we employ to define concurrent graph transformation.

Definition 1 (Concurrent graph transformation approach). A concurrent graph transformation approach is a system $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \mathcal{X}, \mathcal{C})$ the components of which are the following.

- $\mathcal{G}$ is a class of *graphs*.
- $\mathcal{R}$ is a class of *graph transformation rules* such that every $r \in \mathcal{R}_*$ specifies a binary relation on graphs $SEM(r) \subseteq \mathcal{G} \times \mathcal{G}$, which is subject to the following *true-concurrency condition*: $(G, G'') \in SEM(r_1 + r_2)$ for $r_1, r_2 \in \mathcal{R}_*$ implies $(G, G') \in SEM(r_1)$ and $(G', G'') \in SEM(r_2)$ for some $G' \in \mathcal{G}$. Moreover, we assume that $(G, H) \in SEM(\mathbf{0})$ implies $G = H$ for the null element $\mathbf{0} \in \mathcal{R}_*$.
- $\mathcal{X}$ is a class of *graph class expressions* such that each $x \in \mathcal{X}$ specifies a set of graphs $SEM(x) \subseteq \mathcal{G}$.
- $\mathcal{C}$ is a class of *control conditions* such that each $c \in \mathcal{C}$ specifies a set of graph pairs $SEM_{\mathcal{AR},P}(c) \subseteq \mathcal{G} \times \mathcal{G}$ for every set $\mathcal{AR} \subseteq \mathcal{R}_*$ and every set $P \subseteq \mathcal{R}$.

Remarks.

1. The multisets of rules in $\mathcal{R}_*$ are called *parallel rules*. A pair of graphs $(G, G') \in SEM(r)$ for some $r \in R_*$ is an application of the parallel rule r to G with the result G'. It may be also called a direct parallel derivation or a parallel derivation step denoted by $G \underset{r}{\Longrightarrow} G'$. Accordingly, a sequence of parallel derivation steps $G = G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_k}{\Longrightarrow} G_k = G'$ for $k \in \mathbb{N}$ defines a parallel derivation (of length k) from G to G', which may be denoted by $G \overset{*}{\underset{P}{\Longrightarrow}} G'$ if $r_1, \cdots, r_k \in P_*$ for $P \subseteq \mathcal{R}$.
2. The control conditions are meant to restrict the nondeterminism of rule applications so that some reference is needed to the rules in consideration. Hence, the semantics of control conditions has two rules parameters which are used in the definition of the semantics of autonomous units in the next section. On one hand, the semantics of a control condition c of an autonomous unit may depend on the rule set P of the unit itself. On the other hand, it depends on a set $\mathcal{AR}$ of *active rules* typically being the set of parallel rules that can be composed of the rules of all units in a community. Hence, the set $\mathcal{AR}$ specifies all parallel derivations that can be constructed with the rules of a community, and the rule set P indicates which of these rules belong to the unit with the control condition c.

3. Due to the true-concurrency condition, each direct parallel derivation $d = (G \underset{r_1+r_2}{\Longrightarrow} G'')$ gives rise to a parallel derivation $G \underset{r_1}{\Longrightarrow} G' \underset{r_2}{\Longrightarrow} G''$ which is called a *sequentialization* of d. The two parallel derivation steps are called *independent* (of each other). Note that there is a second sequentialization of the form $G \underset{r_2}{\Longrightarrow} \hat{G} \underset{r_1}{\Longrightarrow} G''$ because of the commutativity in R_*. A parallel derivation step and its sequentialization can be considered as *equivalent* w.r.t. true concurrency. If this relation is closed under sequential composition of parallel derivations as well as reflexivity, symmetry, and transitivity, one gets the *true-concurrency equivalence* on parallel derivations, which is denoted by $\equiv$. This is made explicit in the following definition.

Definition 2 (True-concurrency equivalence). Let $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \mathcal{X}, \mathcal{C})$ be a concurrent graph transformation approach, and let $DER(\mathcal{A})$ be the set of all parallel derivations over $\mathcal{A}$. Then the *true-concurrency-equivalence* is recursively defined on $DER(\mathcal{A})$ as follows:

1. Let $d = (G \underset{r_1+r_2}{\Longrightarrow} G'')$ be a direct parallel derivation and let
$$d' = (G \underset{r_1}{\Longrightarrow} G' \underset{r_2}{\Longrightarrow} G'')$$
be its sequentialization. Then $d \equiv d'$.
2. Let $d = (G \overset{*}{\underset{P}{\Longrightarrow}} \overline{G})$, $d' = (G \overset{*}{\underset{P}{\Longrightarrow}} \overline{G})$, $c = (F \overset{*}{\underset{P}{\Longrightarrow}} G)$, and $e = (\overline{G} \overset{*}{\underset{P}{\Longrightarrow}} H)$ be parallel derivations for some $P \subseteq \mathcal{R}$. If $d \equiv d'$, then $c \circ d \equiv c \circ d'$ and $d \circ e \equiv d' \circ e$.[1]
3. $d \equiv d$ for all $d \in DER(\mathcal{A})$.
4. If $d \equiv d'$, then $d' \equiv d$ for all $d, d' \in DER(\mathcal{A})$.
5. If $d \equiv d'$ and $d' \equiv d''$, then $d \equiv d''$ for all $d, d', d'' \in DER(\mathcal{A})$.

For technical simplicity we assume in the following that $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \mathcal{X}, \mathcal{C})$ is an arbitrary but fixed concurrent graph transformation approach.

Examples

In the following we present some instances of the components of concurrent graph transformation approaches which are partly used in the examples of the following sections. Further examples of graph transformation approaches can be found in, e.g., [18].

Graphs. A well-known instance for the class $\mathcal{G}$ is the class of all directed edge-labeled graphs as defined in Section 2. Other classes of graphs are trees, undirected graphs, hypergraphs, etc.

Rules. As a concrete example of rules we consider the so-called DPO rules each of which consists of a triple $r = (L, K, R)$ of graphs such that $L \supseteq K \subseteq R$ (cf. [3]). The application of a rule to a graph G yields a graph G', if one proceeds according to the following steps:

[1] Here, $\circ$ denotes the sequential composition of derivations.

1. Choose a graph morphism $g: L \to G$ so that for all items x, y (nodes or edges) of L with $x \neq y$, $g(x) = g(y)$ implies that x and y are in K.
2. Delete all items of $g(L) - g(K)$ provided that this does not produce dangling edges. (In the case of dangling edges the morphism g cannot be used.) The resulting graph is denoted by D.
3. Add R to the graph D.
4. Glue D and R by identifying the nodes and edges of K in R with their images under g.

The conditions of (1) and (2) concerning g are called gluing condition.

Graph transformation rules can be depicted in several forms. In the following they are shown by drawing only its left-hand side L and its right-hand side R together with an arrow pointing from L to R, i.e. $L \to R$. The nodes of K are distinguished by different shapes and fill-styles occurring in L and R as well.

Figure 1 shows an example of a rule. To interpret the drawing properly, one should remember that loops are not drawn, but the labels of the loops are placed next to the nodes which the loops are incident with. In particular, a v-node is a node with a v-labeled loop and a $\{b, e\}$-node a node with two loops labeled with b and e, respectively. The left-hand side of the rule in Figure 1 consists of a v-node, the intermediate graph K is equal to the left-hand side, and the right-hand side consists of the same v-node, a new $\{b, e\}$-node, and a new edge labeled with v. The edge points from the v-node to the $\{b, e\}$-node. Hence, the rule can be applied to a graph with a v-node with the effect that a new $\{b, e\}$-node and a v-labeled edge from the v-node to the $\{b, e\}$- node are generated.

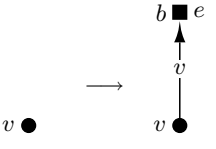


Fig. 1. A rule

Given two rules $r_i = (L_i, K_i, R_i)$ $(i = 1, 2)$, their parallel composition yields the rule $r_1 + r_2 = (L_1 + L_2, K_1 + K_2, R_1 + R_2)$ where $+$ denotes the disjoint union of graphs. In the same way, one can construct a parallel rule from any multiset $r \in \mathcal{R}_*$. For every pair $(G, G'') \in SEM(r_1 + r_2)$ there exists a graph G' such that (G, G') is in $SEM(r_1)$ and (G', G'') is in $SEM(r_2)$. This means that the applications of DPO rules are truly concurrent. (see, e.g., [3] for more details).

Graph class expressions. Every subset $M \subseteq \mathcal{G}$ is a graph class expression that specifies itself, i.e. $SEM(M) = M$. A single graph G can also serve as a graph class expression specifying all graphs G' with $G \subseteq G'$. This type of graph class expressions is called *subgraph condition*. Moreover, every set $\mathcal{L}$ of labels specifies the class of all graphs in $\mathcal{G}$ the labels of which are elements of $\mathcal{L}$. Every set $P \subseteq \mathcal{R}_*$ of (parallel) graph transformation rules can also be used as a graph

class expression specifying the set of all graphs that are reduced w.r.t. P where a graph is said to be reduced w.r.t. P if no rules of P can be applied to the graph. The least restrictive graph class expression is the term *all* specifying the class $\mathcal{G}$.

Control conditions. The least restrictive control condition is the term *free* that allows all pairs of graphs, i.e. $SEM_{\mathcal{AR},P}(free) = \mathcal{G} \times \mathcal{G}$ for all $\mathcal{AR} \subseteq \mathcal{R}_*$ and all $P \subseteq \mathcal{R}$. This is the only control condition used in our running example. A much more restrictive control condition is a single rule $r \in P$. Its semantics $SEM_{\mathcal{AR},P}(r)$ consists of the initial and final graphs of all parallel derivations $G_0 \underset{r_1}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ $(n > 0)$ where $r_1, \ldots, r_n \in \mathcal{AR}$, exactly one r_i contains at least one copy of r, and no other rule of P is applied.[2] More formally, this is expressed as follows:

- There is an $i \in \{1, \ldots, n\}$ such that $r_i(r) > 0$.
- For $j = 1, \ldots, n$, $r_j(r') = 0$, for all $r' \in P \setminus \{r\}$.
- For all $j \in \{1, \ldots, n\}$ with $j \neq i$, $r_j(r) = 0$.

A more general control condition is a set $M \subseteq P$ where $SEM_{\mathcal{AR},P}(M)$ specifies the initial and terminal graphs of all parallel derivations $G_0 \underset{r_1}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ such that for $i = 1, \ldots, n$, $r_i \in \mathcal{AR}$, and $r_i(r) = 0$ if $r \in P \setminus M$. This means that in the applied parallel rules no copy of a rule in $P \setminus M$ may occur.

Since the semantics of control conditions are binary relations, they can be sequentially composed. For control conditions c and c', their sequential composition is denoted by $c\,;c'$ with $SEM_{\mathcal{AR},P}(c\,;c') = SEM_{\mathcal{AR},P}(c) \circ SEM_{\mathcal{AR},P}(c')$. Other useful control conditions are regular expressions, *as long as possible*, as well as priorities (cf. [14]).

4 Communities of Autonomous Units

Autonomous units interact in a common environment which is modeled as a graph. As a basic modeling device, an autonomous unit consists of a set of graph transformation rules, a control condition, and a goal. The graph transformation rules contained in an autonomous unit *aut* specify all transformations the unit *aut* can perform. Such a transformation comprises for example a movement of the autonomous unit within the current environment, the exchange of information with other units via the environment, or local changes of the environment. The control condition regulates the application process. For example, it may require that a sequence of rules be applied as long as possible. The goal of a unit is a graph class expression determining how the transformed graphs should look like eventually.

[2] We assume that identical rules of different autonomous units can be distinguished in $\mathcal{AR}$. This can be achieved by considering named rules. For technical simplicity, this is not further regarded in this paper.

Definition 3 (Autonomous unit). An *autonomous unit* is a system $aut = (g, P, c)$ where $g \in \mathcal{X}$ is the *goal*, $P \subseteq \mathcal{R}$ is a set of graph transformation rules, and $c \in \mathcal{C}$ is a control condition. The components of aut are also denoted by g_{aut}, P_{aut}, and c_{aut}, respectively.

An autonomous unit modifies an underlying environment while striving for its goal. In the setting of a concurrent graph transformation approach, its semantics consists of a set of equivalence classes of parallel derivations w.r.t. the true-concurrency equivalence. This concerns parallel derivations which comprise the parallel application of local rules of the considered unit as well as of rules performed by other autonomous units that are working in the same environment. In a concurrent setting, environment changes performed by other units must be possible while a single autonomous unit applies its rules. To cover this in the definition of the semantics, we assume a variable set of active rules that descibes all possibilities of coexisting units. Moreover, autonomous units regulate their transformation processes by choosing only those rules that are allowed by their control condition. A transformation process is called successful if its last environment satisfies the goal of the unit.

Definition 4 (Parallel and concurrent semantics).

1. Let $aut = (g, P, c)$ be an autonomous unit, let $\mathcal{AR} \subseteq \mathcal{R}_*$ be a set of parallel rules, called *active rules*, and let $d = (G \overset{*}{\underset{\mathcal{AR}}{\Longrightarrow}} G') \in DER(\mathcal{A})$ be a parallel derivation over $\mathcal{A}$. Then d is a *parallel run* of aut if $(G, G') \in SEM_{\mathcal{AR},P}(c)$.
2. The set of parallel runs of aut is denoted by $PAR_{\mathcal{AR}}(aut)$.
3. The derivation d is called a *successful parallel run* if $G' \in SEM(g)$.
4. Let $d \in PAR_{\mathcal{AR}}(aut)$. Then $[d]$ is a *concurrent run* of aut where $[d]$ denotes the equivalence class of d w.r.t. the true-concurrency equivalence $\equiv$, i.e., $[d] = \{d' \mid d \equiv d'\}$.
5. The set of concurrent runs of aut is denoted by $CONCUR_{\mathcal{AR}}(aut)$.
6. A concurrent run $[d]$ is *successful* if it contains a successful parallel run.

Remarks.

1. A parallel run of an autonomous unit depends on its rules and its control condition as the pair of the start graph and the result graph must be accepted by the control condition semantics with respect to the rules of the unit. Moreover, it depends on the set $\mathcal{AR}$ of active rules that reflects the context of the considered unit. The definition of the parallel and concurrent semantics does not fix the set AR. This means that one can choose any set of parallel rules as active. Nevertheless, as mentioned before, the typical case is to choose $\mathcal{AR}$ as the set of all parallel rules composed of the rules of a set of units that interact with each other in the common environment. The rules of the considered unit may occur as single rules or as components of parallel rules in $\mathcal{AR}$.

2. All parallel and concurrent runs contain only derivations that apply active rules. Moreover, each such derivation is only accepted if its initial and result graph are allowed by the control condition. Moreover, each concurrent run contains at least one accepted parallel run. Hence, the set of concurrent runs is the quotient set of the parallel runs with respect to the true equivalence relation. This is reflected in the following observation the proof of which is straightforward and hence omitted.

Observation 1. Let $aut = (g, P, c)$ be an autonomous unit, let $\mathcal{AR} \subseteq \mathcal{R}_*$, and let $d = (G \overset{*}{\underset{\mathcal{AR}}{\Longrightarrow}} G')$ be a derivation. Then the following statements are equivalent.

1. $(G, G') \in SEM_{\mathcal{AR},P}(c)$
2. $d \in PAR_{\mathcal{AR}}(aut)$
3. $[d] \cap PAR_{\mathcal{AR}}(aut) \neq \emptyset$
4. $[d] \in CONCUR_{\mathcal{AR}}(aut)$

Examples

Two examples of autonomous units are depicted in Fig. 2. Both contain a single rule that – according to the control condition *free* – can be applied arbitrarily often. The goal of both units is equal to *all* which means that all parallel and concurrent runs of the units are successful.

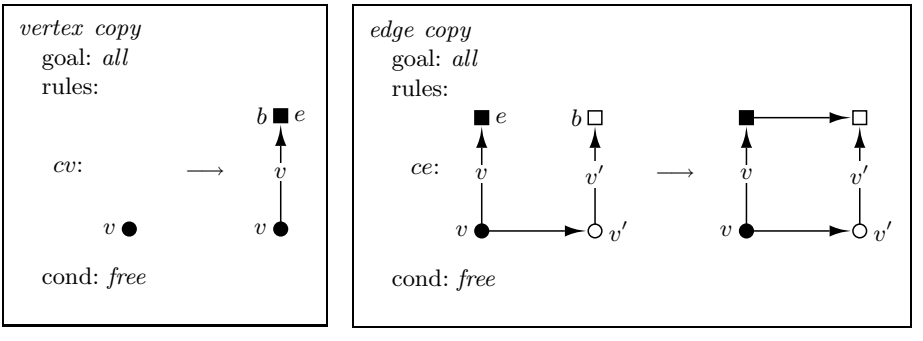


Fig. 2. The units *vertex copy* and *edge copy*

Given a directed edge-labeled graph G and a set V such that every node in G is a v-node, for some $v \in V$, the unit *vertex copy* on the left-hand side of Fig. 2 copies a node of G by generating a $\{b, e\}$-node and a v-labeled edge from the original node to its newly generated copy. The unit *edge copy* on the right side copies one unlabeled edge e' of G provided that $s(e')$ and $t(e')$ are already copied by two executions of *vertex copy*. It erases the e-loop at the copy of $s(e')$ and the b-loop at the copy of $t(e')$ in order to guarantee that b and e cannot be used

by other applications of the rule *ce*. The label b indicates that the corresponding node is the beginning of a simple path and the label e indicates the end of such a path, respectively. Hence, b is removed if a new edge ends at that node, and e is removed if a new edge starts at that node. Multiple concurrent applications of *vertex copy* and *edge copy* generate simple paths from b-nodes to e-nodes that are copies of simple paths of G. Moreover, the units generate copies of simple cycles of G.

Applications of both rules depend on each other only in some cases. Concretely, an application of *ce* is causally dependent on the two copies of its nodes and on any edge copy that tries to use the same b- or e-loop. All other cases are independent so that all vertex copies can be done in parallel followed by all edge copies in parallel in the extreme case.

Autonomous units are meant to work within a community of autonomous units that modify the common environment together. Every community is composed of an overall goal that should be achieved, an environment specification that specifies the set of initial environments the community may start working with, and a set of autonomous units. The overall goal may be closely related to the goals of the autonomous units in the community. Typical examples are the goals admitting only graphs that satisfy the goals of one or all autonomous units in the community.

Definition 5 (Community). A *community* is a triple

$$COM = (Goal, Init, Aut),$$

where $Goal, Init \in \mathcal{X}$ are graph class expressions called the *overall goal* and the *initial environment specification*, respectively, and Aut is a set of autonomous units. The components of COM are denoted as $Goal_{COM}$, $Init_{COM}$, and Aut_{COM}, respectively.

In a community, all units act on the common environment in a self-controlled way by applying their rules. The active rules integrated in the semantics of autonomous units make it possible to define a concurrent semantics of a community in which every autonomous unit may perform its transformation processes. From the point of view of a single autonomous unit, the changes of the environment that are not caused by itself must be activities of the other units in the community. Hence, in every transformation step in a community, a multiset of the rules occurring in the autonomous units of the community is applied to the environment. All these multisets constitute the active rules of the community. This is reflected in the following definition.

Definition 6 (Active rules). Let $COM = (Goal, Init, Aut)$ be a community. Then the set of its *active rules* is defined by

$$\mathcal{AR}(COM) = (\bigcup_{aut \in Aut} P_{aut})_*.$$

Every concurrent run of a community must start with a graph specified as an initial environment of the community. Moreover, it must be a concurrent run of every autonomous unit participating in the community. Successful runs of communities are defined analogously to the successful runs of autonomous units.

Definition 7 (Concurrent community semantics).

1. Let $COM = (Goal, Init, Aut)$ be a community of autonomous units. Let $d_{aut} \in PAR_{\mathcal{AR}(COM)}(aut)$ for every $aut \in Aut$ with $d_{aut} \equiv d_{aut'}$ for all $aut, aut' \in Aut$. Let the common start graph of these equivalent derivations be in $SEM(Init)$. Then the common equivalence class is a concurrent run of COM.
2. The set of all concurrent runs of COM is denoted by $CONCUR(COM)$.
3. A concurrent run is *successful* if the common result graph is specified by $SEM(Goal)$.

As the definition of the community semantics shows, there is a strong connection between the semantics of a community $COM = (Goal, Init, Aut)$ and the semantics of an autonomous unit $aut \in Aut$. The concurrent semantics of COM is a subset of the semantics of aut with respect to the active rules of COM. Conversely, one may take the intersection of the concurrent runs of all autonomous units with respect to the active rules and restrict it to the derivations starting in an initial environment. Then one gets the concurrent semantics of the community. This reflects the autonomy because no unit can be forced to do anything that is not admitted by its own control. The following observation makes the described connection precise.

Observation 2. Let $COM = (Goal, Init, Aut)$ be a community. Then

$$CONCUR(COM) = \{[G \underset{\mathcal{AR}(COM)}{\overset{*}{\Longrightarrow}} G'] \in \bigcap_{aut \in Aut} CONCUR_{\mathcal{AR}(COM)}(aut) \mid G \in Init\}.$$

Proof. Let $d = (G \underset{\mathcal{AR}(COM)}{\overset{*}{\Longrightarrow}} G')$ be a derivation. Then by Definition 7, the class $[d]$ is in $CONCUR(COM)$ if and only if $G \in SEM(Init)$ and for each $aut \in Aut$, there is a derivation $d_{aut} \in PAR_{\mathcal{AR}(COM)}(aut)$ such that $d_{aut} \equiv d$, i.e., $[d_{aut}] = [d]$. By Definition 4, this means that $[d] \in \{[G \underset{\mathcal{AR}(COM)}{\overset{*}{\Longrightarrow}} G'] \in \bigcap_{aut \in Aut} CONCUR_{\mathcal{AR}(COM)}(aut) \mid G \in Init\}$.

If the control conditions satisfy certain properties, the preceding definitions establish a nice relation between the community semantics and the semantics of the single autonomous unit which is composed of the goal of the community, the union of all rules occurring in the autonomous units of the community and a control condition that, specifies the intersection of the semantics of the control conditions of the units. This is stated in the following observation.

Observation 3. Let $COM = (Goal, Init, Aut)$ be a community and let $union = (Goal, \bigcup_{aut \in Aut} P_{aut}, c)$ such that

$$SEM_{\mathcal{AR}(COM),P_{union}}(c) = \bigcap_{aut \in Aut} SEM_{\mathcal{AR}(COM),P_{aut}}(c_{aut}).$$

Then $CONCUR_{\mathcal{AR}(COM)}(union) = \bigcap_{aut \in Aut} CONCUR_{\mathcal{AR}(COM)}(aut)$.

Proof. Let $d = (G \overset{*}{\underset{\mathcal{AR}(COM)}{\Longrightarrow}} G')$ be a derivation. By Observation 1, we have that $[d] \in CONCUR_{\mathcal{AR}(COM)}(union)$ if and only if $(G, G') \in SEM_{\mathcal{AR}(COM),P}(c)$. By assumption, this is the case if and only if

$$(G, G') \in \bigcap_{aut \in Aut} SEM_{\mathcal{AR}(COM),P_{aut}}(c_{aut}),$$

and by Observation 1, this is equivalent to

$$[d] \in \bigcap_{aut \in Aut} CONCUR_{\mathcal{AR}(COM)}(aut).$$

It should be noted that the condition in Observation 3 can be satisfied if the following holds.

1. For each $aut \in Aut$ the semantics of the control condition c_{aut} does not depend on the parameter P_{aut}, i.e.,

$$SEM_{\mathcal{AR}(COM),P}(c_{aut}) = SEM_{\mathcal{AR}(COM),P'}(c_{aut}),$$

   for all $P, P' \subseteq \mathcal{R}$.
2. The class $\mathcal{C}$ is closed under intersection, i.e., for all $c, c' \in \mathcal{C}$ we have $c \wedge c' \in \mathcal{C}$ with $SEM_{\mathcal{AR},P}(c \wedge c') = SEM_{\mathcal{AR},P}(c) \cap SEM_{\mathcal{AR},P}(c')$.

In this case, the condition c of the unit *union* can be defined as $\bigwedge_{aut \in Aut} c_{aut}$.

Examples

The community *V-paths* shown in Fig. 3 solves a generalized form of the Hamiltonian path problem. More precisely, for some set $V = \{v_1, \ldots, v_r\}$, *V-paths* searches for copies of V-paths which are simple paths, that consist of r nodes and contain exactly one v_i-node for every $i = 1, \ldots, r$. Hence, these paths are Hamiltonian w.r.t. the set V and, consequently, if V coincides with the node set of the initial environment, the community solves the Hamiltonian path problem. This is done in a concurrent way with the four autonomous units *V-checker*, *vertex copy*, *edge copy*, and *check*. The initial component of the community specifies the set of *V-graphs* comprising all directed graphs with unlabeled edges and nodes, but where to each node a v-loop is added for some $v \in V$ The goal of *V-paths* is a subgraph condition specifying the set of all graphs that contain a *heureka*-node indicating that a V-path is found.

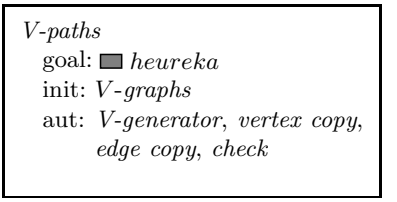


Fig. 3. The community *V-paths*

The autonomous unit *V-checker* generates a set of $(V \cup \{check\})$-nodes where $check \notin V$. This node is used later on for analysing copies of simple paths. It is depicted in Fig. 4. The nodes generated by *V-checker* will be called checkers in the following.

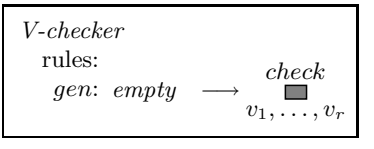


Fig. 4. The unit *V-checker*

The autonomous units *vertex copy* and *edge copy* are depicted in Fig. 2. As stated in Section 4, they copy simple paths of the initial graph and label their start nodes with a *b*-loop and their end nodes with an *e*-loop, each.

The unit *check* is depicted in Fig. 5 and contains the rules *start*, *go*, and *stop*. It searches for a copy of a simple path of the initial environment. The rule *start* begins the search at a *b*-node. It inserts a *go*-edge from the *b*-node to a $(V \cup \{check\})$-checker and deletes both the *v*-loop and the *check*-loop from the latter. The *check*-loop is deleted to avoid that the checker can be used for another path, and the *v*-loop is deleted in order to remember that the path has already passed through a copy of a *v*-node. The application of the rule *go* changes the source of the *go*-edge to the next node, say n, in the copy of a simple path. Moreover, it deletes the corresponding loop at the checker of the path, i.e., it deletes that loop at the target of the *go*-edge which has the same label as the loop of the node of which n is a copy. Hence, the rule *go* cannot move the source of the *go*-edge to an already visited node. The rule *stop* can be applied if there doesn't remain any labeled loop at the corresponding checker. Moreover the copy of the path ends which is indicated by the *e*-loop. The application of the rule deletes the *go*-edge and adds a *heureka*-loop to its target. Please note that the symbol λ at the checker in the left-hand side of the rule means that the checker has no labeled loop. Technically, this can be expressed by the rule with negative context condition (NC, L, K, R) where L consists of an *e*-node, a checker, and a *go*-edge from the *e*-node to the checker, NC consists of L plus a *v*-loop at the

checker where $v \in V$, K is obtained from L by deleting the *go*-edge, and R is obtained from K by adding a *heureka*-loop to the checker.

The control condition is satisfied if the unit *check* applies in its last step a parallel rule composed of the rule *stop* only. Before the application of *stop*, parallel rules composed of *start* and *go* can be applied arbitrarily often. Hence, the unit finishes transforming graphs after the first application of the rule *stop*.

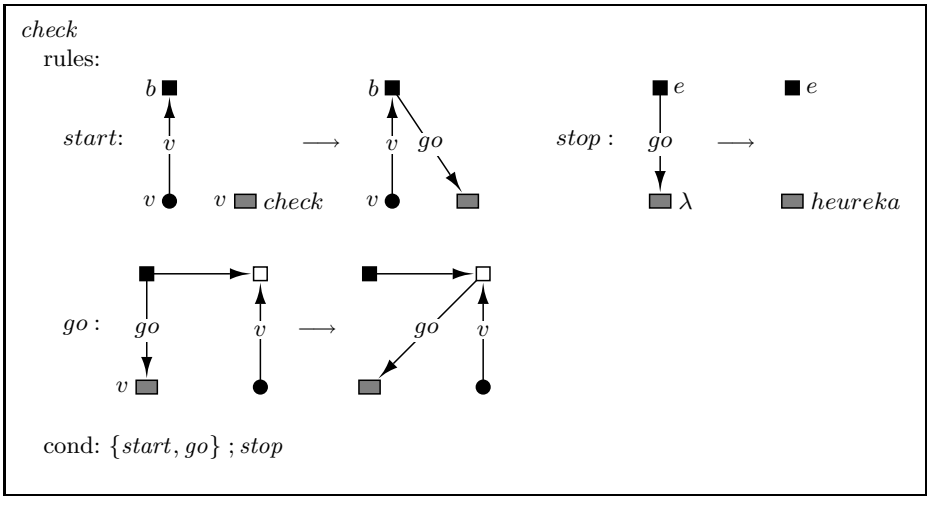


Fig. 5. The unit *check*

With respect to causal dependency, the following holds. The application of the rule of *V-checker* is independent of all other rule applications. The application of *start* must wait for the generation of its node copy and its checker. The rule *go* can be applied after a *start* application sequentially traversing a copy of a simple path. The rule *stop* is only applicable at the end of the path if at all. The applications of the *check*-rules are independent of each other if they concern different copies of simple paths. In particular, the check of a single copy of a simple path is never longer than r steps if r is the number of elements of V. It needs exactly r steps in the successful case of finding a V-path. Because of the control condition of the autonomous unit *check*, we get that all concurrent runs of the community *V-paths* are successful.

5 Canonical Derivations

The definition of concurrent runs of autonomous units reflects the principle of true concurrency meaning that the order of time of two rule applications is only fixed if the derivation steps are causally dependent. The disadvantage of the equivalence classes as concurrent runs is that they may contain an exponential number of equivalent derivations. This follows from the fact that a parallel derivation step applying n rules is equivalent to all iterated sequentializations

including all $n!$ permutations of the n rule applications. There is a complete enumeration of each equivalence class starting with a parallel run in the class by iterated sequentializations and inverse sequentializations. But one may ask whether there is a more efficient method to check the equivalence of parallel runs. How and how fast equivalence can be checked, is often a fundamental question. In the case of concurrent runs of autonomous units, it is of particular interest because an equivalence class of parallel runs satisfies the control condition conditions if one member does. Therefore, the satisfaction of control conditions can only be checked up to equivalence.

In this section, we show that each equivalence class of parallel runs contains canonical derivations which are reduced forms with respect to a shift operator and which can be constructed from an arbitrary run by a quadratic number of shifts at most. A shift is a composition of a sequentialization followed by an inverse sequentialization moving some part of a parallel derivation step to the preceding step. In other words, one gets a quadratic equivalence check in this way if the canonical derivation is a unique representative of a concurrent run. Furthermore, it turns out that the canonical derivation is unique if the shift operator is confluent which applies in the DPO approach for example. For technical purposes, we also introduce the delay of a parallel derivation as the sum of the numbers of steps each atomic rule must wait before it is applied as well as the number of applied atomic rules.

The following notions and results can be found in [11] (cf. also [3]) for the DPO approach to graph transformation. They are adapted here to the case of concurrent runs in communities of autonomous units.

Definition 8 (Shift operator).

1. A parallel derivation $F \underset{p+q}{\Longrightarrow} G' \underset{r}{\Longrightarrow} H$ with $p \neq \mathbf{0} \neq q$ is the *shift* of the parallel derivation $F \underset{p}{\Longrightarrow} G \underset{q+r}{\Longrightarrow} H$ if there is a derivation $F \underset{p}{\Longrightarrow} G \underset{q}{\Longrightarrow} G' \underset{r}{\Longrightarrow} H$ such that the first two steps are the sequentialization of $F \underset{p+q}{\Longrightarrow} G'$ and the last two steps the sequentialization of $G \underset{q+r}{\Longrightarrow} H$.
2. The shift operator is closed under sequential composition of parallel derivations, i.e., a parallel derivation $E \overset{*}{\underset{P}{\Longrightarrow}} F \underset{p+q}{\Longrightarrow} G' \underset{r}{\Longrightarrow} H \overset{*}{\underset{P}{\Longrightarrow}} I$ is a *shift* of the parallel derivation $E \overset{*}{\underset{P}{\Longrightarrow}} F \underset{p}{\Longrightarrow} G' \underset{q+r}{\Longrightarrow} H \overset{*}{\underset{P}{\Longrightarrow}} I$ if $F \underset{p+q}{\Longrightarrow} G' \underset{r}{\Longrightarrow} H$ is a shift of $F \underset{p}{\Longrightarrow} G \underset{q+r}{\Longrightarrow} H$.
3. A parallel derivation is *canonical* if no shift is possible.
4. Let $s = (G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_m}{\Longrightarrow} G_m)$ be a parallel derivation with $r_i = \sum_{j=1}^{n_i} r_{ij}$, $r_{ij} \in \mathcal{R}, n_i \geq 1$ for $i = 1, \cdots, m$. Then the *delay* of s is defined by

$$delay(s) = \sum_{i=1}^{m} (i-1) \cdot n_i$$

and the *number of atomic rules* applied in s by $nar(s) = \sum_{i=1}^{m} n_i$.

Theorem 1. Let $s = (G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_m}{\Longrightarrow} G_m)$ be a parallel derivation with $r_i = \sum_{j=1}^{n_i} r_{ij}$, $r_{ij} \in \mathcal{R}, n_i \geq 1$ for $i = 1, \cdots, m$. Let s' be a shift of s. Then the following holds.

1. $s \equiv s'$.
2. $delay(s') < delay(s)$.
3. $0 \leq delay(s) \leq \frac{n \cdot (n-1)}{2}$ for $n = nar(s)$.
4. Let $s = s_0, s_1, \cdots, s_k$ be a sequence of parallel derivations such that s_i is a shift of s_{i-1} for $i = 1, \cdots, k$. Then $k \leq delay(s)$.
5. The equivalence class $[s]$ contains canonical derivations where some of them are obtained by iterating shifts as long as possible starting with s.
6. The canonical derivation in $[s]$ is unique if the shift operator is confluent, i.e., if s_1 and s_2 are shifts of some parallel derivation s_0, then there is a parallel derivation s_3 which is obtained from s_1 and s_2 by iterated shifts.
7. Let the canonical derivation in $[s]$ be unique. Then $s \equiv \overline{s}$ for some parallel derivation $\overline{s}$ if and only if iterated shifts as long as possible starting in s and $\overline{s}$ yield the same result.

Proof. 1. A shift is a composition of a sequentialization and an inverse sequentialization and yields equivalent derivations, therefore.
2. A shift moves at least one atomic rule to the preceding step such that it is delayed one step less while no rule must wait longer than before.
3. As the shift decreases the delay, a sequentialization increases it. Therefore, the worst case is a purely sequential derivation with one applied atomic rule per step. The delay of such a derivation is the sum of the first $n-1$ natural numbers which is $\frac{n \cdot (n-1)}{2}$.
4. As the delay cannot be negative and decreases with each shift, $delay(s)$ is an upper bound of the number of iterated shifts starting in s.
5. Accordingly, the iteration of shifts as long as possible terminates always yielding a canonical derivation.
6. It is well-known that a relation yields unique results by iterated application if it is confluent and terminating.
7. Let $\hat{s}$ be the result of the iterated shifts starting in s and $\overline{s}$. Then $s \equiv \hat{s} \equiv \overline{s}$ according to Point 1. Conversely, there is only one canonical derivation in $[s]$.

Examples

Let $[s]$ be a successful concurrent run of the community *V-paths*. Without loss of generality, one can assume that s is canonical (otherwise shifts as long as possible yield one). Due to the dependency analysis, all node copies and all checker generations are done in the first step and all edge copies and applications of *start* are done in the second step. The following $r-2$ steps are parallel applications of the rule *go* where r is the number of elements of the label set V. The last step consists of all parallel applications of the rule *stop* which is applied at least once because this is the only way to end successfully. This means that a

successful run finds a V-path (and hence a Hamiltonian path as a special one) in $r + 1$ steps.

If enough node and edge copies are made in a fair way, then one gets copies of all simple paths. If enough checkers are generated, then it can be checked whether there is a V-path among all simple paths. In other words, the concurrent semantics of *V-paths* can solve the V-path problem in a linear number of steps with high probability depending on the number of copies. To make this true, one must assure that concurrent steps are not delayed for too long. The canonical derivation does the job because all possible rule applications are performed as early as possible. But less strict runs will also work if the number of shifts that transform them into the canonical derivation is small.

This result is quite significant as it shows that an *NP*-complete problem can be solved by a parallel run in a polynomial number of steps. This holds for concurrent runs, too, if there is not much unnecessary delay. Clearly, it is well-known that parallelism is a way to overcome the *P=NP*-problem. The message here is that autonomous units do the job if they are many enough and interact properly. To keep the example simple, we do not employ any kind of heuristics. Nevertheless, autonomous units are suitable candidates to model heuristic methods (see e.g. [20,15].)

6 Related Work

The present investigation is mainly related to work in three areas. With respect to the concurrent semantics, it contributes to the *theory of concurrency*. Petri nets are shown to be special cases of communities of autonomous units (cf. [9]). The relation to other models of concurrent processes is still an open research topic.

Concerning autonomy, our approach is closely related to *multiagent systems*. In [19], it is sketched that communities of autonomous units are a kind of rule-based realization of the axiomatic definition of multiagent systems in the sense of Wooldridge and others (see,e.g., [21]). How our concept is related to other graph-transformational approaches to agent and actor systems like [10,5,4], should be investigated in the future.

Last but not least, communities of autonomous units are devices to model *interactive processes and distributed systems*. Within the area of graph transformation, the approach is closely related to Manfred Nagl's and others' work on the IPSEN Project and the IMPROVE Project (see, e.g., [16,17]). While we start from a theoretical base and try to expand the concepts to reach practical use, the Nagl school is rooted in the software engineering point of view from the beginning. It would be of great interest to investigate the relations in more details because quite some synergy may emerge from a common framework.

7 Conclusion

This is the third paper on the semantics of autonomous units. After the sequential semantics in [8] and the parallel semantics in [13], we have introduced the

concurrent semantics based on the idea of true concurrency. A concurrent run is an equivalence class of parallel runs w.r.t. the true-concurrency equivalence. It can be represented by canonical derivations where a derivation is canonical if no shift is possible meaning that each rule application is in the first step or causally dependent of the preceding step. The example shows that an *NP*-hard problem can be solved by linear concurrent runs of a community of autonomous units where a concurrent run is linear if it contains a parallel run of linear length.

The paper provides the very first investigation of the concurrent semantics of autonomous units. Further studies are needed to get a better insight into the matter. This includes a thorough comparison with other approaches to concurrency like communicating sequential processes, calculus of communicating systems, traces, bigraphs, etc. W.r.t. Petri nets, the relation is already quite clear. In [13], it is shown that a place/transition system can be seen as a community of autonomous units where each transition is the single rule of a unit and the firing of a multiset of transitions satisfies the true-concurrency condition so that these communities of place/transition systems fit into the framework of this paper. Moreover, it should be thoroughly studied what are of the consequences of requiring that an equivalence class must contain a single successful run.

References

1. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A., Taentzer, G.: Graph transformation for specification and programming. Science of Computer Programming 34(1), 1–54 (1999)
2. Baldan, P., Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Rossi, F.: Concurrent semantics of algebraic graph transformations. In: Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallelism, and Distribution, vol. 3, pp. 107–185. World Scientific, Singapore (1999)
3. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In: Rozenberg [18], pp. 163–245
4. Depke, R., Heckel, R.: Modeling and analysis of agents' goal-driven behavior using graph transformation. In: Ryan, M.D., Meyer, J.-J.C., Ehrich, H.-D. (eds.) Objects, Agents, and Features. LNCS, vol. 2975, pp. 81–97. Springer, Heidelberg (2004)
5. Depke, R., Heckel, R., Küster, J.: Formal agent-oriented modeling with graph transformation. Science of Computer Programming 44, 229–252 (2002)
6. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
7. Hölscher, K., Klempien-Hinrichs, R., Knirsch, P., Kreowski, H.-J., Kuske, S.: Autonomous units: Basic concepts and semantic foundation. In: Hülsmann, M., Windt, K. (eds.) Understanding Autonomous Cooperation and Control in Logistics – The Impact on Management, Information and Communication and Material Flow, pp. 103–120. Springer, Heidelberg (2007)

8. Hölscher, K., Kreowski, H.-J., Kuske, S.: Autonomous units and their semantics — the sequential case. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 245–259. Springer, Heidelberg (2006)
9. Hölscher, K., Kreowski, H.-J., Kuske, S.: Autonomous units to model interacting sequential and parallel processes. Fundamenta Informaticae 92(3), 233–257 (2009)
10. Janssens, D.: Actor grammars and local actions. In: Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallelism, and Distribution, vol. 3, pp. 57–106. World Scientific, Singapore (1999)
11. Kreowski, H.-J.: Manipulationen von Graphmanipulationen. Ph.D. thesis, Technische Universität Berlin (1977)
12. Kreowski, H.-J., Kuske, S.: Graph transformation units with interleaving semantics. Formal Aspects of Computing 11(6), 690–723 (1999)
13. Kreowski, H.-J., Kuske, S.: Autonomous units and their semantics - the parallel case. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 56–73. Springer, Heidelberg (2007)
14. Kuske, S.: More about control conditions for transformation units. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 323–337. Springer, Heidelberg (2000)
15. Kuske, S., Luderer, M.: Autonomous units for solving the capacitated vehicle routing problem based on ant colony optimization. Electronic Communications of the EASST 26, 23 pages (2010)
16. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996)
17. Nagl, M., Marquardt, W. (eds.): Collaborative and Distributed Chemical Engineering: From Understanding to Substantial Design Process Support - Results of the IMPROVE Project. LNCS, vol. 4970. Springer, Heidelberg (2008)
18. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific, Singapore (1997)
19. Timm, I., Kreowski, H.-J., Knirsch, P., Timm-Giel, A.: Autonomy in software systems. In: Hülsmann, M., Windt, K. (eds.) Understanding Autonomous Cooperation and Control in Logistics – The Impact on Management, Information and Communication and Material Flow, pp. 255–274. Springer, Heidelberg (2007)
20. Tönnies, H.: An evolutionary graph transformation system as a modelling framework for evolutionary algorithms. In: Mertsching, B., Hund, M., Aziz, Z. (eds.) KI 2009. LNCS, vol. 5803, pp. 201–208. Springer, Heidelberg (2009)
21. Wooldridge, M., Jennings, N.R.: Intelligent agents: Theory and practice. The Knowledge Engineering Review 10(2) (1995)

Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation

Enrico Biermann[1], Hartmut Ehrig[1], Claudia Ermel[1], Ulrike Golas[1], and Gabriele Taentzer[2]

[1] Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
{ehrig,ugolas,lieske,enrico}@cs.tu-berlin.de
[2] Fachbereich Mathematik und Informatik
Philipps-Universität Marburg, Germany
taentzer@mathematik.uni-marburg.de

Abstract. The theory of algebraic graph transformation has proven to be a suitable underlying formal framework to reason about the behavior of model transformations. In order to model an arbitrary number of actions at different places in the same model, the concept of amalgamated graph transformation has been proposed. Rule applications of certain regularity are described by a rule scheme which contains multi-rules modeling elementary actions and a common kernel rule for their synchronization (amalgamation). The amalgamation theorem by Böhm et al. ensures that for two multi-rules, the application of the amalgamated rule yields the same result as two iterative rule applications, respecting their common kernel rule application. In this paper, we propose an extension of the amalgamation theorem to an arbitrary finite number of synchronous rule applications. The theorem is used to show parallel independence of amalgamated graph transformations by analyzing the underlying multi-rules. As example, we specify an excerpt of a model transformation from Business Process Models (BPM) to the Business Process Execution Language (BPEL).

1 Introduction

Model transformation is one of the key activities in model-driven software development. The theory of algebraic graph transformation has proven to be a suitable underlying formal framework to reason about the behavior of model transformations, e.g. showing termination and confluence [1].

Although graph transformation is an expressive, graphical and formal means to describe computations on graphs, single rules are not always sufficient to express complex actions. In that case, a specification of either sequential or parallel rule application is needed to define a potentially infinite set of rule apllications in a finite way. Often, the problem arises to model an arbitrary number of similar parallel actions at different places in the same model. One

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 121–140, 2010.

way to transform multiple recurring structures (multi-object structures) is the sequential application of rules by explicitly encoding an iteration over all the actions to be performed. Often, this solution is not as declarative as it could be, since we have to care about the stepwise execution of a *forall* operation on a collection.

Hence, we propose the use of *amalgamated graph transformation* concepts, based on parallel graph transformation introduced in [2] and extended to synchronized transformations in [3,4], to define model transformations with multi-object structures. The essence of amalgamated graph transformation is that (possibly infinite) sets of rules with some regularity can be described by a finite set of *multi-rules* modeling the elementary actions, amalgamated at a kernel rule. The synchronization of rules along kernel rules in so-called *interaction schemes* allows a transformation step to be maximally parallel in the following sense: An amalgamated rule, induced by an interaction scheme, is constructed by a number of multi-rules being synchronized at the kernel rule. The number of multi-rules is determined by the number of different multi-rule matches found such that they all overlap in the match of the kernel rule. Hence, the transformation of multi-object structures can be described in a general way though the number of actually occurring objects in the instance model is variable.

The *amalgamation theorem* by Böhm et al.[3] ensures that for two instances of a multi-rule the application of the amalgamated rule yields the same result as the iterative application of these two rules, where their common kernel is transformed only once. In this paper, we propose an extension of the amalgamation theorem to an arbitrary finite number of multi-rules. Using this extension, the transformation of models with variably many multi-object structures can be handled in a general way. An additional result allows to analyze parallel independence of amalgamated graph transformations by analysing the underlying multi-rules. The concept of amalgamation is useful for several kinds of model transformation between visual models. In this paper, we present the theoretical results without proofs, while the theory with proofs is presented in the categorical framework of weak adhesive HLR categories in [5].

The main aim of this paper is to show how amalgamated graph transformation can be applied to model transformation. For this purpose, the theoretical results are applied to a model transformation translating simple business process models written in the Business Process Modeling Notation (BPMN) to executable processes formulated in the Business Process Execution Language for Web Services (BPEL). BPMN models allow to split the control flow in an arbitrary number of branches which are rejoined later. An analogous transformation step is the translation of fork-join structures. We use this excerpt of the model translation to motivate our amalgamation concept. The independence result of this paper can be used to check if several applications of these amalgamated transformation steps are independent of each other and can be executed in parallel.

We assume that the reader is familiar with the basic concepts of algebraic graph transformation and basic constructions like pushouts and pullbacks which are used in Sections 3 and 4. For an introduction to these concepts we refer to [1].

The paper is structured as follows: In the next section, our running example, an excerpt of a model transformation from BPMN to BPEL, is introduced. Amalgamated graph transformation is reviewed in Section 3 where also the new extended amalgamation theorem is presented. This theorem is used in Section 4 to show parallel independence of amalgamated graph transformations. Finally, Section 5 considers related approaches (including tool-based approaches) and Section 6 concludes the paper.

2 Example: Model Transformation from BPMN to BPEL by Amalgamated Graph Transformation

In this section, we describe our running example, an excerpt of a model transformation from the Business Process Modeling Notation (BPMN) to the Business Process Execution Language for Web Services (BPEL) according to the translation by Van der Aalst et al. in [6].

BPMN is a standard notation for modeling business processes used for business analysis. In addition, BPEL is introduced as a notation to define executable processes, i.e. processes that can be loaded into an execution engine which will then coordinate the activities specified in the process definitions. Since both languages are quite complex, we will consider subsets of the BPMN and BPEL notations only. In addition, we use a reference structure to connect BPMN elements with their newly constructed BPEL counterparts. During the transformation, the BPMN is modified as well because structures are collapsed (similiar to parsing) while being transformed to BPEL.

In this paper, we concentrate on the translation of BPMN *And* and *Xor* constructs to the corresponding BPEL language elements *Flow* and *Switch* via amalgamated graph transformations. Translating those constructs with ordinary graph transformation rules would require a complex control structure for guiding the rule application process. Other BPMN language constructs like *While* or *Repeat* can be handled by normal graph transformation rules. The complete model transformation case study is described in [7] from a practical point of view. The model transformation is implemented using AGG [8], a development environment for attributed graph transformation systems supporting the algebraic approach to graph transformation [1]. AGG has been extended recently by support for defining and applying amalgamated graph transformation. All screenshots in this paper are taken from the AGG editors for rules and interaction schemes.

2.1 Type Graphs

We define type graphs of the source language BPMN and the target language BPEL. Furthermore, in order to define transformation rules, relations between source and target meta-models are given by reference nodes of type F2ARef. The type graph integrating the BPMN source model (left-hand part), the reference

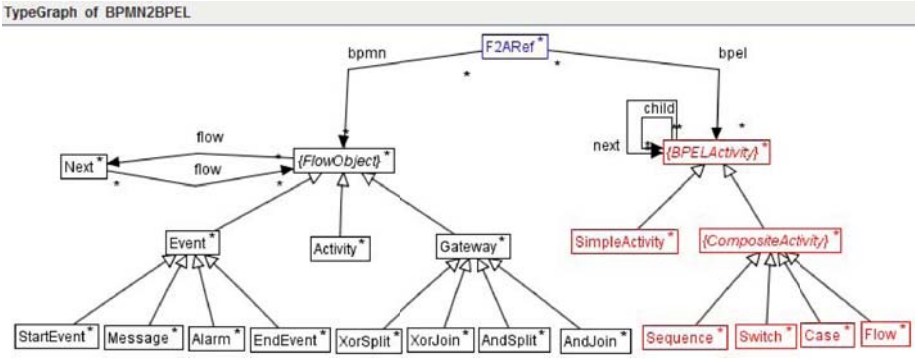


Fig. 1. BPMN2BPEL type graph

part (the node type F2ARef and its adjacent edge types bpmn and bpel) and the target model (right-hand part) is shown in Fig. 1.

For the BPEL type graph, the parent-child relation on activities has been modeled as a child relation. This leads to a structure similar to the XML representation of the BPEL language in [6].

2.2 A BPMN Instance Graph: ATM Machine

As an example we consider a BPMN diagram which models a person's interaction with an ATM (see Fig. 2 where the concrete and abstract syntax of the diagram are depicted). In the upper part, the ATM machine accepts and holds the card of the person while simultaneously contacting the bank for the account information. (The language elements AndSplit and AndJoin are used to model parallel actions.) Afterwards, the display prompts the user for the PIN. Depending on the user's input there are three possible outcomes:

- the user enters the correct PIN and can withdraw money,
- the user enters the wrong PIN (a message is displayed),
- the user aborts the operation (an alarm signal is given).

For modelling alternative actions, the language elements XOrSplit and XOrJoin are used. Any further interaction with the ATM (like returning the card to the user) is not modelled here. Next nodes without conditions have an empty string as "*cond*" attribute. For simplicity, in Fig. 2 (b), these Next nodes are shown without their attributes.

In the following, we will use the abstract syntax only where the source BPMN model like the ATM machine in Fig. 2, the corresponding target BPEL model and all intermediate models of the model transformation are typed over the type graph in Fig. 1.

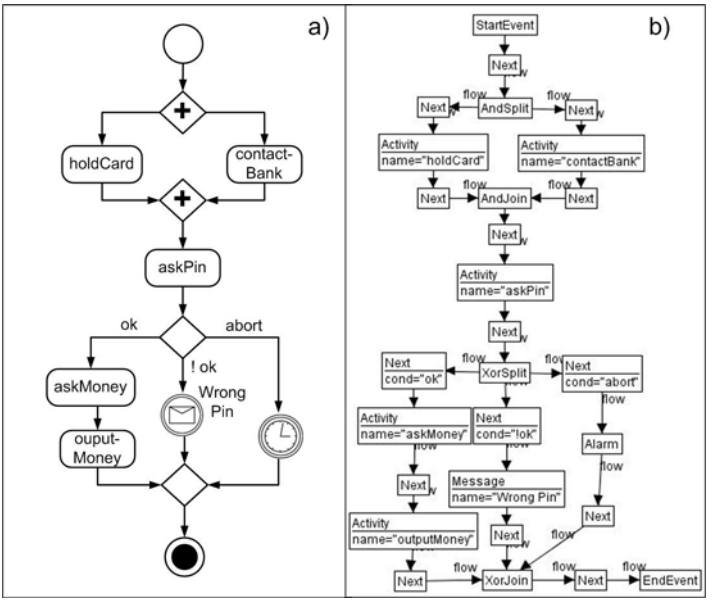


Fig. 2. ATM machine in BPMN in concrete syntax (a) and abstract syntax (b)

2.3 Transformation Rules

Basically, the transformation from BPMN to BPEL is modelled in two phases:

1. translate all simple activities from BPMN to BPEL,
2. collapse all complex structures to new activities in the source model while creating the corresponding structure in the target model, e.g. *sequences*, *switch* or *flow* structures.

Rule *CreateSimpleActivity* in Fig. 3 models the translation of a simple activity from BPMN to BPEL. Note that the negative application condition (NAC) takes care that a corresponding BPEL SimpleActivity is created only if its BPMN Activity has not already been translated before. Similar rules model the translation of *Alarm* and *Message* events which are changed to an activity with a corresponding *SimpleActivity* as translation. This eases the succeeding steps of the translation, because these events are handled like activities in BPEL.

While translating structures from BPMN to BPEL, reference nodes are created connecting source model flow objects to BPEL activities in the target model.

The translation of a chain of BPMN activities is done by two rules shown in Figs. 4 and 5: Rule *CreateSequence* creates a new sequence in the BPEL model and inserts the first two activities into this sequence. Rule *CreateSequence2* deals with the case that the sequence has already been created in the BPEL model.

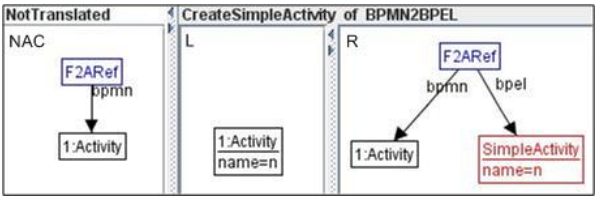


Fig. 3. Rule *CreateSimpleActivity*

Then, the current activity is added to the existing sequence. The negative application condition (NAC) of each rule ensures that activities in a sequence are translated from top to bottom, i.e. there are no other untranslated activities in the sequence above the current ones, and that the order from the BPMN structure is preserved in the BPEL model. Note that we do not show all NACs in the figures, but only a representative.

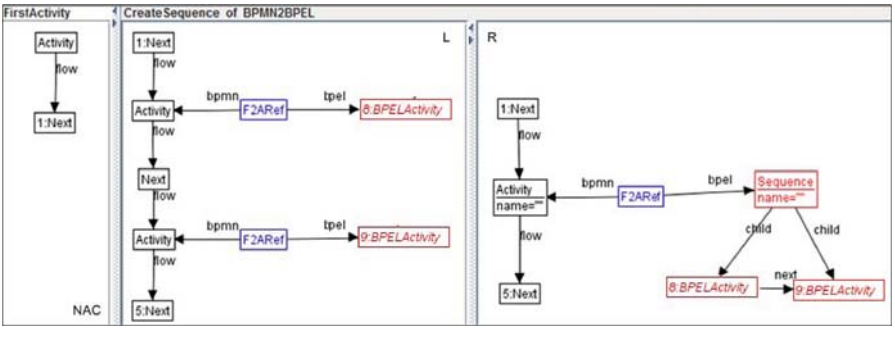


Fig. 4. Rule *CreateSequence*

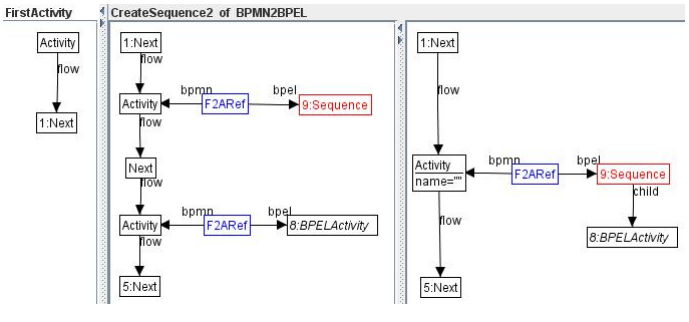


Fig. 5. Rule *CreateSequence2*

In Fig. 6, the result after applying rules *CreateSimpleActivity* and *CreateSequence* as long as possible to the BPMN model in Fig. 2 is shown where all activities and (simple) sequences are translated into the corresponding BPEL objects. The next step is to translate the And and Xor constructs.

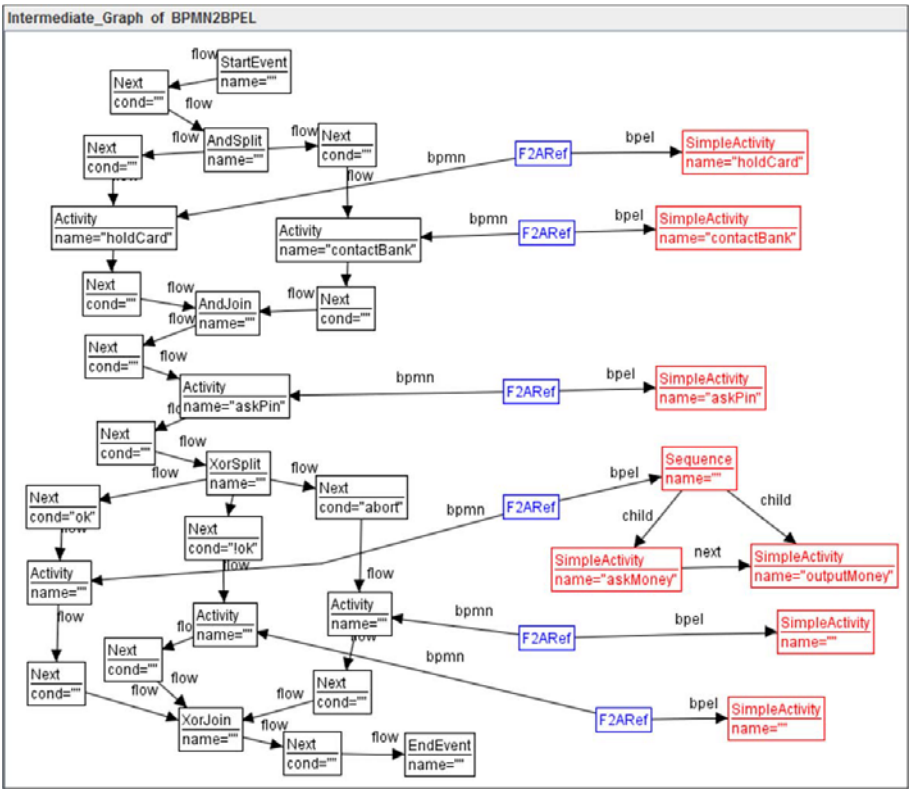


Fig. 6. Intermediate result of the model transformation

An And construct (a number of branches surrounded by an AndSplit and AndJoin element) is translated to a Flow container node which contains a child for each branch emerging from the AndSplit. Since the number of branches can be arbitrary, a normal graph transformation rule or any finite number of rules would not be sufficient to express this situation. Therefore, we use amalgamated graph transformation to express maximal parallel execution in the following sense: A common subaction is modelled by the application of a common *kernel rule* of all additional actions (modelled by *multi-rules*). For example, in order to process an And construct, the common subaction processes one branch only. Independent of the number of additional branches, this is the kernel action which will always

happen. Hence, we model this action by the kernel rule in the upper part of Fig. 7 where one branch surrounded by an AndSplit and AndJoin is translated to a BPEL Flow node with one child. The NAC takes care that there are no other branches with sequences of activities within the And construct (i.e. sequences should be collapsed into single activities before translating the And construct).

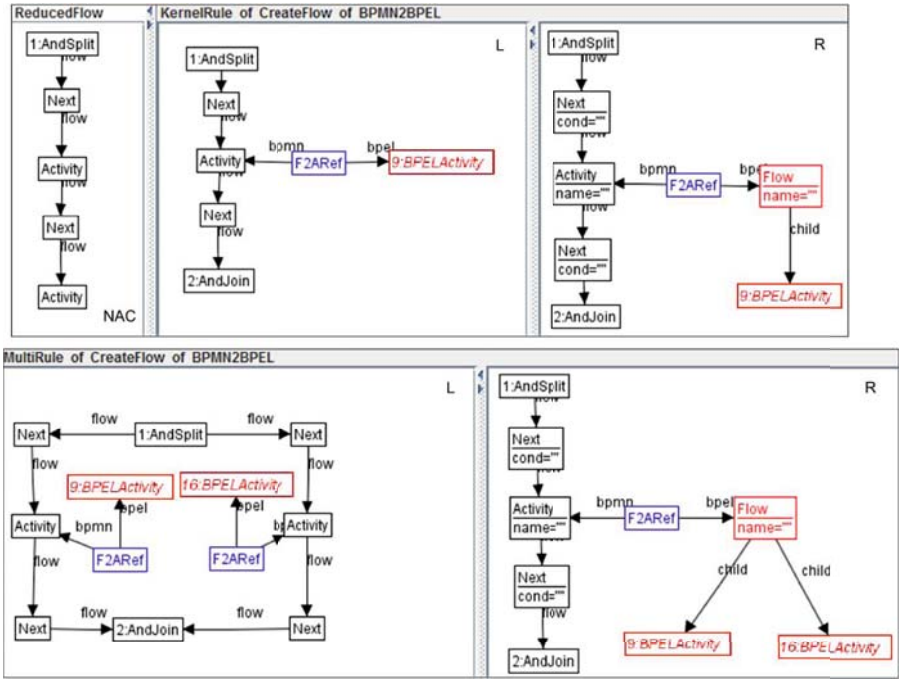


Fig. 7. Interaction scheme *CreateFlow*

Additional subactions now are modelled by multi-rules. Each multi-rule contains at least the kernel rule and specifies an additional action which is executed as many times as matches for this additional part can be found in the host graph. The multi-rule for processing And constructs is shown in the bottom part of Fig. 7. It extends the kernel rule by one more branch. Formally, the synchronization possibilities of kernel and multi-rules are defined by an *interaction scheme* consisting of a kernel rule and a set of rules called multi-rules such that the kernel rule is embedded in each of the multi-rules. The kernel morphism from the kernel rule to the multi-rule is indicated in Fig. 7 by single node mappings shown by corresponding numbers of some of the graph objects. Note that all missing node and edge mappings can be uniquely determined from the given ones.

The application of an interaction scheme, i.e. of multi-rules synchronized at their kernel rule is twofold: At first, a match of the kernel rule is selected. Then, multi-rule instances are constructed, one for each new match of a multi-rule in the current host graph such that it overlaps with the kernel match only. Then, all multi-rule instances are glued at their corresponding kernel rule objects which leads to a new rule, the *amalgamated rule*. The application of the amalgamated rule at the amalgamated match consisting of all multi-rule matches glued at the kernel match is called *amalgamated graph transformation*.

For the handling of Xor constructs (a number of branches surrounded by an XorSplit and an XorJoin element), we have an analogous interaction scheme *CreateSwitch* which also consists of a kernel rule processing one branch, and one multi-rule taking care of additional branches (see Fig. 8). Here, a BPEL Switch node with corresponding children is created, where the condition in the Next node is translated to a Case distinction.

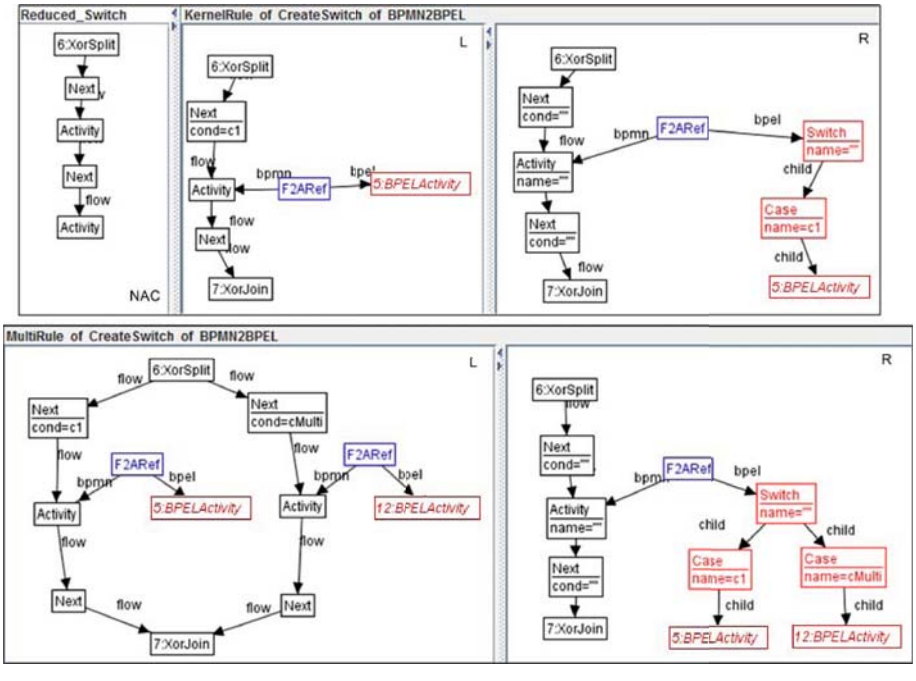


Fig. 8. Interaction scheme *CreateSwitch*

The application of the CreateSwitch interaction scheme to the Xor construct depicted in the bottom part of Fig. 6 yields an amalgamated rule where the kernel rule is glued with two instances of the multi-rule (since we have three branches between the XorSplit and the XorJoin).

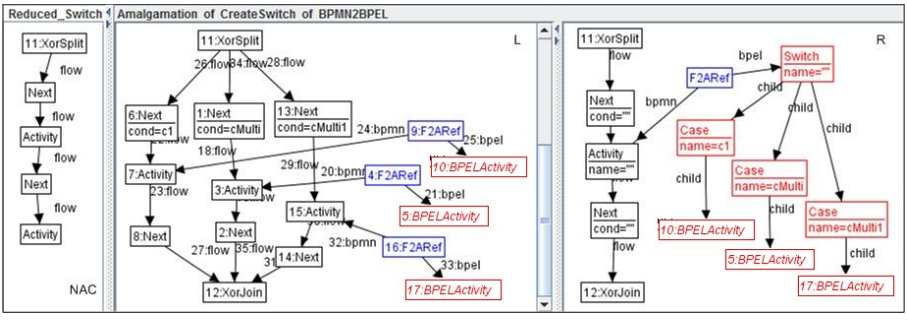


Fig. 9. Amalgamated rule of *CreateSwitch* for the ATM model

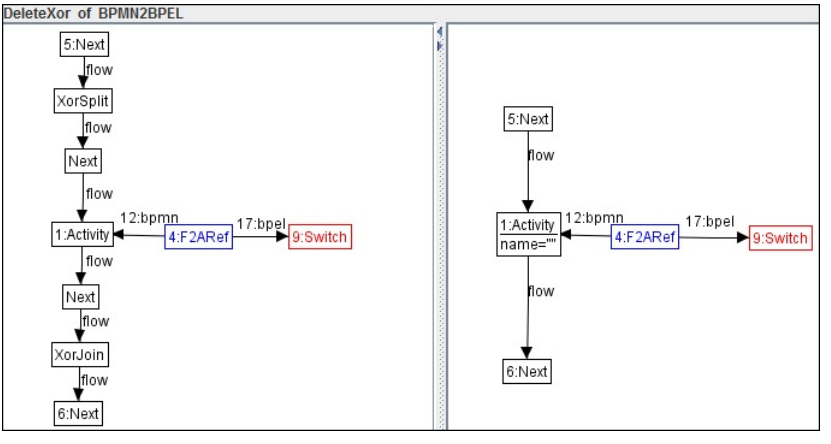


Fig. 10. Rule *DeleteXor*

The amalgamated rule (shown in Fig. 9) is then used to process the three branches in one step by applying it to the graph in Fig. 6. The NAC shown in Fig. 9 originates from the kernel rule and is simply copied from it. NACs of multi-rules would require shifting to preserve their meaning as is explained in [5]. After the application of the amalgamated rule, the rule DeleteXor in Fig. 10 removes the XorSplit and XorJoin which are only connected by a single branch with a single acitivity after the application of the amalgamated rule. A similar rule DeleteAnd removes the AndSplit and AndJoin after the successful Flow construction.

If we apply our transformation rules in a suitable order starting with the BPMN model ATM in Fig. 2, we get the resulting integrated graph shown in Fig. 11 (b). Here, some elements of the BPMN notation still exist, comprising a sort of stop graph of the parsing process because no transformation rule can be applied anymore. If a pure BPEL graph is desired, this remaining BPMN structure can be deleted by an additional rule. The abstract syntax of the BPEL

expression is the tree with root node Sequence which is the target of the bpel edge from the F2ARef node. The XML syntax of the BPEL model corresponding to this tree is shown in Fig. 11 (a).

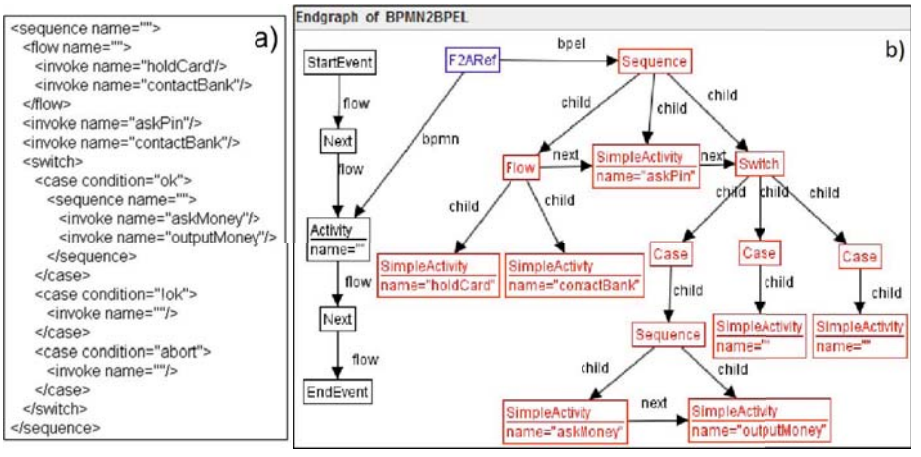


Fig. 11. ATM machine as transformation result in concrete BPEL syntax (a) and in abstract syntax (b)

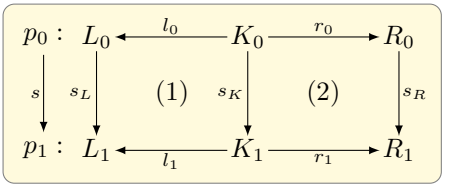
3 Amalgamated Graph Transformation

In this section, we give the formal foundations of amalgamated graph transformation based on graph transformation in the well-known double-pushout approach. We assume the reader to be familiar with this approach (see [1] for an introduction and a large number of theoretical results). We concentrate on the presentation of new concepts and results. The formal proofs of our results can be found in [5]. For simplicity, we present the theory without negative application conditions and without attributes, but it has been extended already to this more general case which is used in our example. In fact, the theory in [5] is presented in the categorical framework of weak adhesive HLR categories [1] for rules with nested application conditions in the sense of Habel-Pennemann [9] where negative application conditions are a special case.

Formally, a kernel morphism describes how the kernel rule is embedded into a multi-rule. We need some more technical preconditions to make sure that the embeddings of the L-, K-, and R-components are consistent.

Definition 1 (Kernel Morphism). *Given rules* $p_0 = (L_0 \xleftarrow{l_0} K_0 \xrightarrow{r_0} R_0)$ *and* $p_1 = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ *with injective morphisms* l_i, r_i *for* $i \in \{0,1\}$, *a* kernel morphism $s : p_0 \to p_1$, $s = (s_L, s_K, s_R)$ *consists of injective morphisms* $s_L : L_0 \to L_1$, $s_K : K_0 \to K_1$, *and* $s_R : R_0 \to R_1$ *such that in the following*

diagram (1) *and* (2) *are pullbacks, and* (1) *has a pushout complement for* $s_L \circ l_0$. p_0 *is then called* kernel rule *and* p_1 multi-rule.



Example 1. The kernel rule and the multi-rule of *CreateFlow* in Fig. 7 are linked by a kernel morphism, which is indicated in Fig. 12 by equal numbering of selected elements. The mapping of all remaining elements can be induced uniquely. Fig. 12 shows that we have pullbacks (1) and (2). A pullback of two injective graph morphisms can be considered as intersection of two graphs embedded in a common graph. E.g. pullback (1) models the intersection K_0 of L_0 and K_1 in L_1. Morphisms s_L and l_1 indicate how K_0 and L_0 are embedded in L_1. The pushout complement for $s_L \circ l_0$ adds the graph part which is needed to complete L_0 over K_0 such that L_1 is the result. Note that the pushout complement is not the graph K_1 in Fig. 12, but looks like the left-hand side of the rule in Fig. 13.

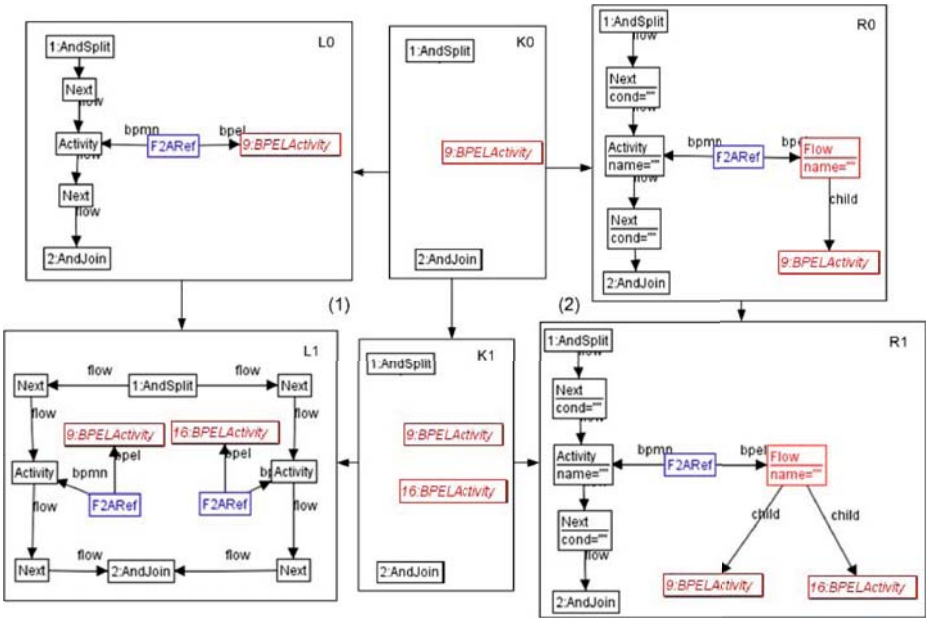


Fig. 12. Kernel morphism of interaction scheme *CreateFlow*

For a given kernel morphism, the complement rule is the remainder of the multi-rule after the application of the kernel rule, i.e. it describes what the multi-rule does in addition to the kernel rule. The following lemma is an important construction which is used in Theorem 1.

Lemma 1 (Complement Rule). *Given rules* $p_0 = (L_0 \xleftarrow{l_0} K_0 \xrightarrow{r_0} R_0)$ *and* $p_1 = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ *and a kernel morphism* $s : p_0 \to p_1$ *then there exists a* complement rule $\overline{p_1} = (\overline{L} \xleftarrow{\overline{l}} \overline{K} \xrightarrow{\overline{r}} \overline{R})$ *such that each transformation* $G \overset{p_1}{\Longrightarrow} H$ *can be decomposed into a transformation sequence* $G \overset{p_0}{\Longrightarrow} G' \overset{\overline{p_1}}{\Longrightarrow} H$.

Proof Idea: Intuitively, the complement rule is the smallest rule that extends K_0 such that it creates and deletes all these parts handles by the multi but not by the kernel rule.

Example 2. For the kernel rule and the multi-rule of *CreateFlow* in Fig. 7, the complement rule is shown in Fig. 13. It handles the transformation of the additional branch which has to be added as an activity to the already existing flow.

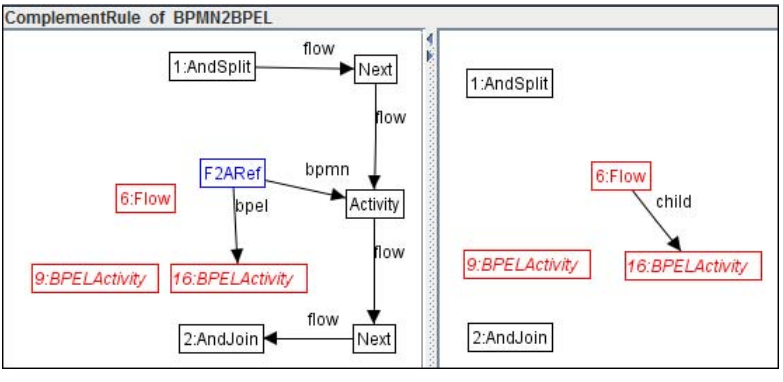


Fig. 13. Complement rule of *CreateFlow*

According to Fact 1, each transformation by the multi-rule of *CreateFlow* (Fig. 7) applied to the graph in Fig. 6 can be decomposed into a transformation sequence applying first the kernel rule of *CreateFlow* (Fig. 7) to the graph in Fig. 6 and then the complement rule of *CreateFlow* in Fig. 13 to the resulting graph.

A bundle of kernel morphisms over a common kernel rule forms an interaction scheme.

Definition 2 (Interaction Scheme). *Given rules* $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$ *for* $i = 0, \ldots, n$, *an* interaction scheme *is a bundle of kernel morphisms* $s = (s_i : p_0 \to p_i)_{i=1,\ldots,n}$.

Given an interaction scheme which describes the basic actions in its kernel rule and a set of multi-rules, we need to construct a graph-specific interaction scheme for a given graph and kernel match. This specific interaction scheme contains a certain number of multi-rule copies for each multi-rule of the basic scheme. To do so, we search all different multi-rule matches which overlap in the kernel match. The number of different multi-rule matches determines how many copies are included in the graph-specific interaction scheme which is the starting point for the amalgamated rule construction defined in Def. 3.

Consider Fig. 14 as an example. The basic interaction scheme is given on the left. It consists of kernel rule r_0 which adds a loop. Moreover, it contains one multi-rule r_1 modeling that object 2 being connected to object 1 is deleted and a new object is created and connected to object 1 which has a loop now. Note that there is a bundle of kernel morphisms which embed the kernel rule into the multi-rules. Given graph G, there are obviously three different matches from multi-rule r_1 to G which overlap in the match of the kernel rule to G. Hence, the multi-rule can be applied three times. Thus, we need three copies of the multi-rule in the graph-specific interaction scheme, all with kernel morphisms from kernel rule r_0. In our example, the graph-specific interaction scheme is shown on the right. Gluing all multi-rules in the graph-specific interaction scheme at their common kernel rule, we get the amalgamated rule with respect to G, shown at the bottom of Fig. 14.

In the following definition, we clarify the construction of amalgamated rules from interaction schemes which are intended to be graph-specific already.

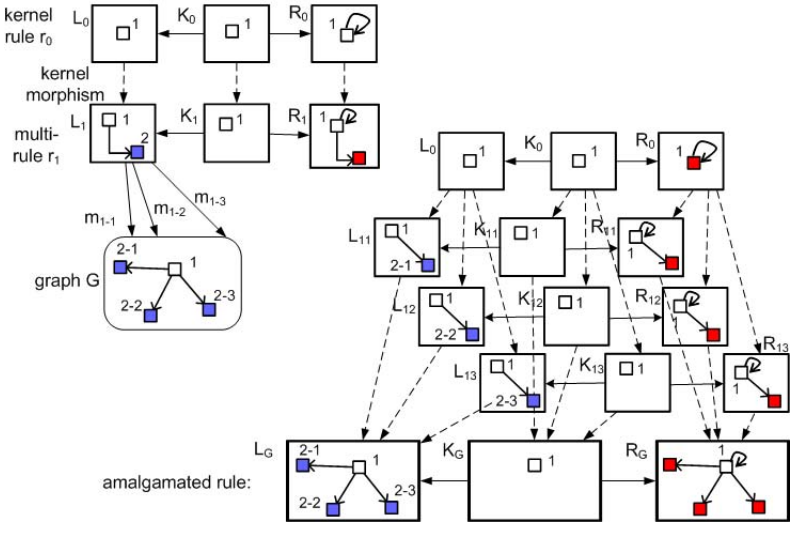
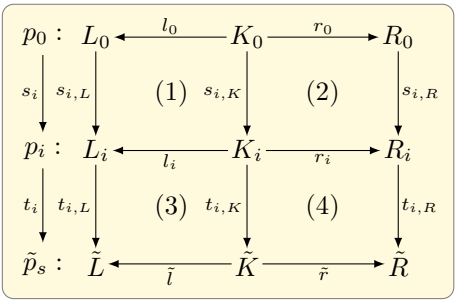


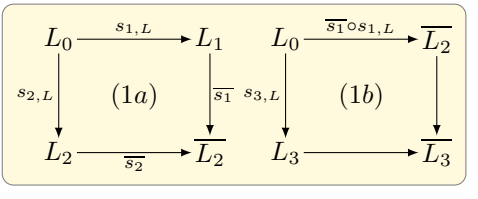
Fig. 14. Construction of an amalgamated rule

Definition 3 (Amalgamated Rule). *Given rules* $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$ *for* $i = 0, \ldots, n$ *and an interaction scheme* $s = (s_i : p_0 \to p_i)_{i=1,\ldots,n}$*, then the* amalgamated rule $\tilde{p}_s = (\tilde{L} \xleftarrow{\tilde{l}} \tilde{K} \xrightarrow{\tilde{r}} \tilde{R})$ *is constructed componentwise over* s_i *as stepwise pushouts over* i*.*



Remark 1. We sketch the idea how to construct the componentwise pushout for $n = 3$ for the L-component:

1. Construct the pushout $(1a)$ of $s_{1,L}$ and $s_{2,L}$.
2. Construct the pushout $(1b)$ of $\overline{s_1} \circ s_{1,L}$ and $s_{3,L}$.
3. $\overline{L_3}$ is the resulting left-hand side $\tilde{L}$ for the amalgamated rule.



This construction is unique, independent of the order of i, and can be done similarly for the K- and R-components. By pushout properties (see [1]), we obtain unique morphisms $\tilde{l}$ and $\tilde{r}$.

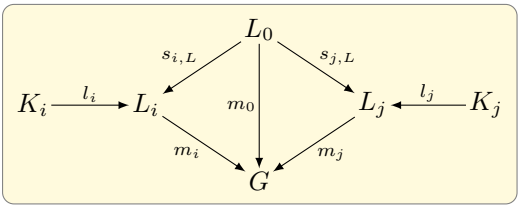
Example 3. In our example transformation for the ATM machine from BPMN in Fig. 2 to BPEL in Fig. 11 we can use two different amalgamated rules - one for the transformation of the And construct and one for the Xor construct. For the And construct, the multi-rule in Fig. 7 is applied once which means that $n = 1$ and the amalgamated rule is isomorphic to the multi-rule itself. For the Xor construct, the same multi-rule is applied twice, leading to the amalgamated rule shown in Fig. 9.

The application of an amalgamated rule to a graph G is called an amalgamated transformation. Since an amalgamated rule is a normal transformation rule, an amalgamated graph transformation step at an amalgamated rule and an amalgamated match is a normal graph transformation step. If we have a bundle of direct transformations of a graph G, where for each transformation one of the multi-rules is applied, we want to analyse if the amalgamated rule is applicable

to G combining all the single transformation steps. These transformations are compatible, i.e. multi-amalgamable, if the matches agree on the kernel rules, and are independent outside.

Definition 4 (Multi-Amalgamable). *Given an interaction scheme* $s = (s_i : p_0 \to p_i)_{i=1,\dots,n}$*, a bundle of direct transformations steps* $(G \overset{p_i,m_i}{\Longrightarrow} G_i)_{i=1,\dots n}$ *is* multi-amalgamable *over* s*, if*

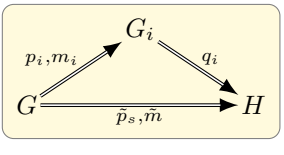
- *it has* consistent matches*, i.e.* $m_i \circ s_{i,L} = m_j \circ s_{j,L} =: m_0$ *for all* $i, j = 1, \dots, n$ *and*
- *it has* weakly independent matches*, i.e.* $m_i(L_i) \cap m_j(L_j) \subseteq m_0(L_0) \cup (m_i(l_i(K_i)) \cap m_j(l_j(K_j)))$ *for all* $1 \leq i \neq j \leq n$ *which means that the elements in the intersection of the matches* m_i *and* m_j *are either preserved by both transformations, or are also matched by* m_0*.*



Remark 2. The concepts of *consistent matches* and *weakly independent matches* have been used already for the case $n = 2$ in [3].

If a bundle of direct transformations of a graph G is multi-amalgamable, then we can apply the amalgamated rule directly to G leading to a parallel execution of all the changes done by the single transformation steps.

Theorem 1 (Multi-Amalgamation). *Given a bundle of multi-amalgamable transformations* $(G \overset{p_i,m_i}{\Longrightarrow} G_i)_{i=1,\dots,n}$ *over an interaction scheme* s *then there is an amalgamated transformation* $G \overset{\tilde{p}_s,\tilde{m}}{\Longrightarrow} H$ *and transformations* $G_i \overset{q_i}{\Longrightarrow} H$ *over some rule* q_i *such that* $G \overset{p_i,m_i}{\Longrightarrow} G_i \overset{q_i}{\Longrightarrow} H$ *is a decomposition of* $G \overset{\tilde{p}_s,\tilde{m}}{\Longrightarrow} H$*. Note that* q_i *is the complement rule of the kernel morphism* $t_i : p_i \to \tilde{p}_s$ *and can be constructed as a gluing of the complement rules* $\overline{p_1}, \dots, \overline{p_n}$ *of* $p_1, \dots, p_n$ *over* K_0*.*



Proof Idea: Basically, $\tilde{p}_s$ is applicable to G because the bundle is multi-amalgamable. Then we can apply Lemma 1 which implies the decomposition.

Example 4. The multi-rule of the interaction scheme *CreateSwitch* (Fig. 8) can be applied two times to the graph in Fig. 6 with the same kernel rule match

(Fig. 8). In fact, the interaction scheme *CreateSwitch* leads to multi-amalgamable transformations $G \overset{p_i,m_i}{\Longrightarrow} G_i, i = 1, 2$. By Theorem 1, we can apply the amalgamated rule (Fig. 9) to the same graph G, leading to $G \overset{\tilde{p}_s,\tilde{m}}{\Longrightarrow} H$, and for $G \overset{p_i,m_i}{\Longrightarrow} G_i$ we have $G_i \overset{q_i}{\Longrightarrow} H$ with $q_1 = \overline{p_2}$ and $q_2 = \overline{p_1}$ where $\overline{p_1}$ and $\overline{p_2}$ are the complement rules of p_1 and p_2, respectively.

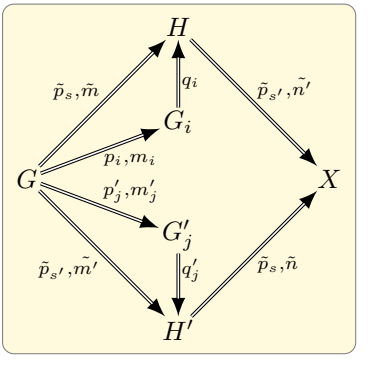
4 Parallel Independence of Amalgamated Graph Transformations

In this section, we want to analyze when two amalgamated graph transformations of a graph are parallel independent which means that they can be sequentially applied in any order. Of course, we could check parallel independence of the two transformations directly based on the definition and using the amalgamated rules. But even if we only know the underlying bundles of transformation steps via the multi-rules, we can analyze parallel independence of the amalgamated transformations based on these single transformation steps.

Theorem 2 (Parallel Independence). *Given two interaction schemes s and s' and two bundles of multi-amalgamable transformations $(G \overset{p_i,m_i}{\Longrightarrow} G_i)_{i=1,\dots,n}$ over s and $(G \overset{p'_j,m'_j}{\Longrightarrow} G'_j)_{j=1,\dots,n'}$ over s' such that $G \overset{p_i,m_i}{\Longrightarrow} G_i$ and $G \overset{p'_j,m'_j}{\Longrightarrow} G'_j$ are parallel independent for all pairs i, j. Then also the corresponding amalgamated transformations $G \overset{\tilde{p}_s,\tilde{m}}{\Longrightarrow} H$ and $G \overset{\tilde{p}_{s'},\tilde{m'}}{\Longrightarrow} H'$ are parallel independent.*

Proof Idea: For parallel independence, there have to exist morphisms from the gluing objects of the one rule to the result of the second transformation and vice versa. If these morphisms exist for the single transformation steps, we can combine them to morphisms from the gluing objects of the amalgamated rules to the resulting objects of the amalgamated transformations.

Moreover, the Local Church–Rosser Theorem leads to an object X with amalgamated transformations $H \overset{\tilde{p}_{s'},\tilde{n'}}{\Longrightarrow} X$ and $H' \overset{\tilde{p}_s,\tilde{n}}{\Longrightarrow} X$.



Example 5. In the intermediate model in Fig. 6, we find a match for the kernel rule of *CreateFlow* (Fig. 7) using the left branch of the And construct, and an

extension of this match for the multi-rule of *CreateFlow* covering also the right branch. This means that we get a bundle consisting of one transformation with the multi-rule of *CreateFlow*.

Similarly, we can find a match for the kernel rule of *CreateSwitch* (Fig. 8) using the left branch of the Xor construct which can be extended to two different matches of the corresponding multi-rule, one using the middle branch and one using the right branch. This leads to a bundle consisting of two transformations with two instances of the multi-rule of *CreateSwitch*.

If we apply all these multi-rules via the above defined matches, we obtain a transformation $G \overset{p_1}{\Longrightarrow} H_1$ handling the transformation of the And construct, and two transformations $G \overset{p_1'}{\Longrightarrow} H_1'$ and $G \overset{p_2'}{\Longrightarrow} H_2'$ handling the transformation of the left and middle resp. left and right branches of the Xor construct. For the first transformation, the multi-rule is isomorphic to the amalgamated rule because we have $n = 1$. Moreover, the last two transformations are multi-amalgamable because they agree on the kernel rule match and the matches are disjoint outside the kernel. This means that we can apply the amalgamated rule of *CreateSwitch* which is depicted in Fig. 9, directly to the graph in Fig. 6 transforming the complete Xor construct.

The used matches of p_1 and p_1' as well as of p_1 and p_2' are disjoint which means that the corresponding transformations are parallel independent. Therefore we can apply Theorem 2 to these amalgamated transformations which means that the amalgamated transformations of the And and the Xor constructs are parallel independent and can be applied in any order to G leading to the same result X.

5 Related Work

There are several graph transformation-based approaches which realize the transformation of multi-object structures. PROGRES [10] and Fujaba [11] feature so-called set-valued nodes which can be duplicated as often as necessary. These two approaches handle multi-objects in a pragmatic way. Object nodes are indicated to be matched optionally once, arbitrarily often, or at least once. Object nodes that are not indicated as mulit-objects are matched exactly once. Adjacent arcs are treated accordingly. These two approaches are more restricted than ours, since they focus on multiple instances of single nodes instead of graph parts.

Furthermore, PROGRES allows to embed the right-hand side of a rule in a flexible way by specifying an embedding clause. This clause considers the embedding of the corresponding left-hand side and declares for each type of arc running between the left-hand side and the context graph how it is replaced by an arc between the right-hand side and the context graph. Such an embedding clause cannot be defined in our approach. Instead we have to specify a multi-rule for each arc type used in the embedding clause. Since the context graph can be an arbitrary one, so can be the embedding. This is reflected by constructing an amalgamated rule suitable for the actual context and embedding.

Further approaches that realize amalgamated graph transformation are AToM^3, GReAT and GROOVE. AToM^3 supports the explicit definition of interaction schemes in different rule editors [12] whereas GROOVE implements rule amalgamation based on nested graph predicates [13]. The GReAT tool can use a group operator to apply delete, move or copy operations to each match of a rule [14].

A related conceptual approach aiming at transforming collections of similar subgraphs is presented in [15]. The main conceptual difference to our approach is that we amalgamate rule instances whereas Grønmo et al. replace all collection operators (multi-objects) in a rule by the mapped number of collection match copies. Similarly, Hoffmann et al. define a cloning operator in [16] where cloned nodes roughly correspond to multi-objects.

None of the aforementioned approaches so far investigate the formal analysis of amalgamated graph transformation. Here, to the best of our knowledge the amalgamation theorem in [3] has been the only formal result up to now. In this paper, we extend the amalgamation theorem to an arbitrary finite number of multi-rules. We implemented our approach in AGG (used for modeling the sample model transformation in this paper), and in our EMF transformation tool EMF Henshin [7,17]. Here, graph transformation concepts are transferred to model transformations in the Eclipse Modeling Framework (EMF).

6 Conclusions and Future Work

In this paper, we recall the concept of amalgamated graph transformations from [3], extend it to multi-amalgamation and apply it to model transformation. In fact, the amalgamation concept is very useful for this application area, because it allows to define graph transformation systems more compactly. Parallel actions which shall be performed on a set of similar object structures can be described by interaction schemes. An amalgamated graph transformation applies an interaction scheme, i.e. a set of parallel actions with the kernel rule as synchronization point, to a specific graph as one atomic step.

Although amalgamated graph transformations are useful for specifying model transformations more naturally and more efficiently, the theory is not fully developed and thus, cannot be used to verify model transformations yet. Our paper can be considered as an essential contribution towards a theory of amalgamated graph transformations. Due to the multi-amalgamation theorem, the central technical result in this paper, we are able to characterize parallel independent amalgamated graph transformations. If we can show that two alternative steps in a model transformation sequence are parallel independent, we can apply them in any order. This result on parallel independence is essential to show confluence results for amalgamated graph transformations in future work. In [5], the theory of amalgamated graph transformation is formulated already in the framework of weak adhesive HLR categories such that it can be applied to typed attributed graph transformations with suitable application conditions. Moreover, we plan to extend the amalgamation concept to triple graph grammars [18] which are well-known for the specification of model transformations.

References

1. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science. Springer, Heidelberg (2006)
2. Ehrig, H., Kreowski, H.J.: Parallel graph grammars. In: Lindenmayer, A., Rozenberg, G. (eds.) Automata, Languages, Development, pp. 425–447. North Holland, Amsterdam (1976)
3. Böhm, P., Fonio, H.R., Habel, A.: Amalgamation of graph transformations: a synchronization mechanism. Computer and System Sciences (JCSS) 34, 377–408 (1987)
4. Taentzer, G.: Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems. PhD thesis, TU Berlin, Shaker Verlag (1996)
5. Golas, U.: Multi-Amalgamation in $\mathcal{M}$-Adhesive Categories. Technical Report 2010/05, Technische Universität Berlin (2010)
6. Ouyang, C., Dumas, M., ter Hofstede, A.H.M., van der Aalst, W.M.P.: From BPMN process models to BPEL web services. In: Proceedings of the International Conference on Web Services (ICWS 2006), pp. 285–292. IEEE Computer Society, Los Alamitos (2006)
7. Biermann, E., Ermel, C.: Transforming BPMN to BPEL with EMF Tiger. In: Proceedings of the Workshop on Graph-based Tools (GraBaTs 2009) (2009)
8. TFS-Group, TU Berlin: AGG (2009), http://tfs.cs.tu-berlin.de/agg
9. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19, 1–52 (2009)
10. Schürr, A., Winter, A., Zündorf, A.: The PROGRES-approach: Language and environment. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2, pp. 487–550. World Scientific, River Edge (1999)
11. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
12. de Lara, J., Ermel, C., Taentzer, G., Ehrig, K.: Parallel graph transformation for model simulation applied to timed transition Petri nets. ENTCS 109, 17–29 (2004)
13. Rensink, A., Kuperus, J.H.: Repotting the geraniums: On nested graph transformation rules. ECEASST 18 (2009)
14. Balasubramanian, D., Narayanan, A., Neema, S., Shi, F., Thibodeaux, R., Karsai, G.: A subgraph operator for graph transformation languages. ECEASST 6 (2007)
15. Grønmo, R., Krogdahl, S., Møller-Pedersen, B.: A collection operator for graph transformation. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 67–82. Springer, Heidelberg (2009)
16. Hoffmann, B., Janssens, D., van Eetvelde, N.: Cloning and expanding graph transformation rules for refactoring. ENTCS 152, 53–67 (2006)
17. Biermann, E., Ermel, C., Taentzer, G.: Lifting parallel graph transformation concepts to model transformation based on the Eclipse modeling framework. ECEASST 26 (2010), http://journal.ub.tu-berlin.de/index.php/eceasst/issue/view/36
18. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)

Extended Triple Graph Grammars with Efficient and Compatible Graph Translators

Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr

Technische Universität Darmstadt, Real-Time Systems Lab,
Merckstr. 25, 64283 Darmstadt, Germany
{felix.klar,marius.lauder,alexander.koenigs}@es.tu-darmstadt.de,
andy.schuerr@es.tu-darmstadt.de
http://www.es.tu-darmstadt.de

Abstract. Model-based software development processes often force their users to translate instances of one modeling language into related instances of another modeling language and vice-versa. The underlying data structure of such languages usually are some sort of graphs. Triple graph grammars (TGGs) are a formally founded language for describing correspondence relationships between two graph languages in a declarative way. Bidirectional graph language translators can be derived from a TGG, which maps pairs of related graph instances onto each other. These translators must fulfill certain compatibility properties with respect to the correspondence relationships established by their TGG. These properties are guaranteed for the original TGG approach as published 15 years ago. However, its expressiveness is pushed to the limit in most real world scenarios. Furthermore, the original approach relies on a parsing algorithm with exponential runtime complexity. In this contribution, we study a more expressive class of TGGs with *negative application conditions* and show for the first time that derived translators with a polynomial runtime complexity still preserve the above mentioned compatibility properties. For this purpose, we introduce a new characterization of well-formed TGGs together with a new translation rule scheduling algorithm that considers *dangling edges* of input graphs.

Keywords: triple graph grammars, bidirectional model transformation, negative application conditions, dangling edge condition.

1 Introduction

Languages in general and computer languages in particular are used to describe artifacts and processes of the real world. These descriptions can be thought of as models of the real world or in a very general term as *data*. Languages have specific rules to form these models and model elements are mostly typed to assign semantics to a category of elements. Specific domains of the world require specific languages which are nowadays often called *domain specific languages* (DSL). These DSLs are tailored to the specific needs of a group of people which should benefit from using these DSLs. But, two languages that refer to the same

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 141–174, 2010.

domain and share the same artifacts of the real world, may though have different representations. There are various reasons for the concurrent usage of different languages in engineering projects. Domain experts (humans or computers) do not "speak" one language but different ones. Or, one language might be more adequate to express domain specific facts than the other. So, it is a common scenario that an engineer manually translates an instance of one language into an instance of another language and spends a lot of time to keep these language instances then in a consistent state. But, translation between languages is time-consuming and requires experts in both languages. Therefore, it is reasonable to facilitate this process. Accordingly, (semi-)automated translators are needed that assist their users in keeping related instances of languages in a consistent state. Due to the reasons listed above, it seems to be quite natural to develop a language for the language translation domain, too. *Triple graph grammars (TGGs)* are a formally founded language that is used to build bidirectional translators easily.

In the following section, we briefly introduce the reader to the world of TGGs and state the challenges the designers of TGG-based languages have to face: (1) TGGs must be expressive enough for the specification of complex relationships between pairs of languages and (2) they must allow for the derivation of efficiently working translators that are compatible with the related TGG specification. Subsequently, we show in Sect. 3 that we can improve the expressiveness of TGGs by introducing *negative application conditions (NACs)* and nevertheless still derive compatible translators for a certain family of TGGs. In Sect. 4 we add the concept of checking *dangling edge conditions (DECs)* to our graph translation algorithms. DEC-checks resolve rule application conflicts, prevent the construction of illegal graphs, and thus guarantee a polynomial runtime behavior of the derived translators. Based on these achievements we present an efficient translation algorithm in Sect. 5 that is still compatible with TGGs of the formerly defined family of TGGs. Section 6 analyzes how related approaches deal with the challenges of bidirectional model transformation languages in general and TGG-based languages in particular. Finally, we conclude this contribution in Sect. 7 and state open challenges to be solved in future work.

2 TGGs with Negative Application Conditions

In 1994 the first publication appeared that introduced the concept of triple graph grammars [1]. It allows for the specification of correspondence relationships between two languages of graphs. TGGs are grammars that generate graph triples by applying productions of the grammar to an input graph triple (axiom). The three graphs are often called *source* and *target graph* representing the elements of the related languages, and *correspondence graph* containing correspondence links. The correspondence links are elements of a language, too—the correspondence language that relates elements of the source and target language. TGGs have been invented to support translation of documents based on related graph-like data structures. Related documents are, e.g., design documents of a piece of software (e.g., class diagrams) and design documents of a data management

environment (e.g., relational database schema) that persists the data which is processed by the software. Class diagrams and database schemata are closely related as we will see later on. TGGs enable to explicitly establish mappings between documents by means of traceability links that contain additional information about the transformation process. TGGs are used to build bidirectional-working formal language translators. In addition, they allow to check consistency of related documents and to efficiently propagate changes in one document to restore consistency in the corresponding document.

In the context of the model driven architecture (MDA) [2,3] TGGs are used for the declarative specification of bidirectional model transformations. One TGG serves as input for the derivation of a pair of model translators that transform a model of one language into a corresponding model of the other language and vice versa. The correspondence relations are needed for traceability purposes and they encode additional information about the translation process itself. This allows for the realization of incremental updates that are required if changes occur in the involved models. In this contribution we will use the terms "graph", "node", etc. of the graph grammar world, but a translation of all definitions and theorems to the MDA world simply requires the replacement of the introduced terms by their MDA counterparts like "model", "object", ... [4].

Now, we will explain TGGs by example and afterwards describe how translators are derived from a TGG. Finally, we will summarize the fundamental properties of TGGs and derived translators as stated in [5]. The example used throughout this contribution is the well-known mapping between class diagrams, also referred to as "source domain", and relational database schemata, also referred to as "target domain". This example is discussed in numerous publications of the model and graph transformation domain [6,3]. We have reduced the example to a core part that is used to explain the ideas presented in this contribution. We will now start building the triple graph grammar TGG_{CDDS} that specifies the mapping between class diagrams and database schemata.

2.1 TGG Schema

TGGs consist of a *TGG schema* which describes the structural dependencies between the elements of the two related languages and the correspondence language. Figure 1 shows the TGG schema of the triple graph grammar TGG_{CDDS}. The schema defines the structural correspondences of the source and target domain. The domain of class diagrams defines classes and attributes. Classes may have subclasses and contain zero to many attributes. Attributes are ordered by the successor/predecessor relationship "Precedes". Multiple inheritance is not supported in this example, so each subclass may have only one superclass. The domain of relational database schemata defines tables and columns. Each table may contain a number of columns. Columns are ordered by the successor/predecessor relationship "Precedes".

The elements of both domains are related via so called *correspondence link types* which are located in the *link domain*. Link types are denoted as hexagonal elements. Throughout this contribution we call instances of link types *(TGG)*

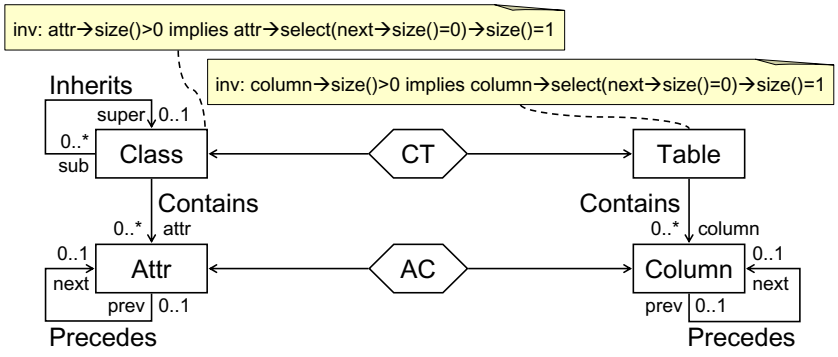


Fig. 1. TGG schema of TGG_{CDDS} that relates class diagrams and database schemata

links. The link type CT (*class-table-relation*) maps classes to tables, whereas the link type AC (*attribute-column-relation*) maps attributes to columns. Let us have a closer look at the graph constraints of the given schema. Graph triples must fulfill these constraints in order to be valid. Successors of attributes and columns are realized via the "Precedes" edge type. The multiplicity "0..1" of the edge type's end "next" denotes that each attribute and column respectively may have a successor, but need not. These are multiplicity constraints that might be expressed by OCL invariants $inv_{CD:P:n:mult}$ and $inv_{DS:P:n:mult}$. In addition, the OCL invariants shown in Fig. 1 constrain the number of elements without successor. A class with attributes must have exactly one attribute without successor ($inv_{CD:C}$). Similarily, tables and columns are constrained by $inv_{DS:T}$.

2.2 TGG Productions

In addition to the TGG schema, a set of *TGG productions* is specified. TGG productions are often called *TGG rules*. However, we stick to the term *TGG productions* to avoid clashes with derived *translation rules*. Productions define a language of consistent graph triples, i.e., they describe how related triples of graphs may evolve simultaneously. Each production consists of a graph pattern—its left-hand side L—that looks for a corresponding match (redex) in a graph triple. Applied to a redex a TGG production adds a copy of the elements of its right-hand side R that are not already part of L to the regarded graph triple.

The TGG productions of TGG_{CDDS} are depicted in Fig. 2 using a shorthand notation for graph productions. Instead of showing both left-hand and right-hand side as two separate parts of a production, both sides are merged [7]. The elements contained in the left-hand and in the right-hand side of the production $L \cap R$ are denoted as black elements without any additional markup. These elements are *context elements* that define a pattern that matches the redex in a host graph triple to which the production is then applied. The elements contained in the right-hand side only $R \backslash L$ are denoted as green elements with an additional $++$ markup. These elements are created during the application of a production

to its redex. Nodes of a TGG production that are created by the production and attached to a TGG link that is created by the production are called *primary nodes*. In general, TGG productions may create additional *secondary nodes* (i.e., *non-primary nodes*) that are directly or transitively connected with the primary node. As the TGG formalism does not allow for the deletion of elements, elements that are contained in the left-hand side only $L \backslash R$ need not to be visualized in TGG productions. Groups of elements that form a NAC are crossed-out. No match for these elements must be found in the host graph; otherwise the matching NAC blocks the application of its production.

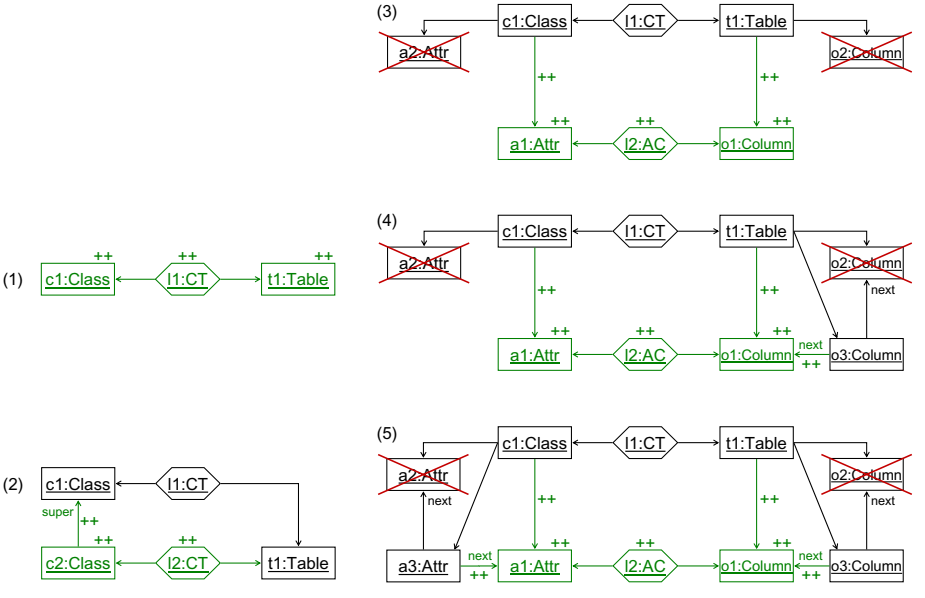


Fig. 2. TGG productions (1) and (2) that relate classes and tables. TGG productions (3), (4), and (5) that relate attributes and columns.

Productions (1) and (2) produce classes and tables, whereas productions (3), (4), and (5) produce attributes and columns. Production (1) is applicable in any situation, as it has no required context elements. It creates a new class in the source domain, a new table in the target domain, and a new CT link in the link domain. Furthermore, the new CT link relates the just created elements of source and target domain. Production (2) creates a new class *c1* and a new CT link *l2* if a class *c1* and a table *t1* exist that are already related via a CT link *l1*. The new class is added to the inheritance structure of *c1* and the new CT link relates *c2* and *t1*. The following productions are more interesting as they make use of NACs, which guarantee that no invalid graph triples are produced by the productions. An invalid graph triple would either violate a multiplicity constraint or an OCL invariant of the TGG schema (cf. Fig. 1).

Production (3) is used to create the first attribute in a class of an inheritance structure and the corresponding column in a table. Due to the NACs present in both domains, this production is applicable only if the matched class and table do not already contain an attribute or a column. So, the NACs ensure that at most one attribute and column are created per class and table respectively that has no successor and, therefore, prohibit a violation of $inv_{CD:C}$ and $inv_{DS:T}$.

Production (4) creates the first attribute of a class of an inheritance structure, when another class of the inheritance structure already has an attribute. In this case the corresponding table already contains at least one column and, as columns are ordered, the new column *o1* must be the successor of one of the existing columns. The NAC in the target domain ensures that the created column *o1* is the successor of a column that does not have a successor yet. So, it ensures that the multiplicity constraint $inv_{DS:P:n:mult}$ of the endpoint "next" in the target domain holds after production application. The NAC in the source domain has the same effect as the NAC in production (3). The new column *o1* has no successor and so enables the production to be applied for other classes that are part of the inheritance structure that have no attribute yet.

Production (5) is applicable in situations where a class and a table have at least one attribute and column respectively, i.e., productions (3) or (4) have been applied earlier on. The effect of both NACs is similar to the NAC of the target domain in production (4). They ensure that the multiplicity constraints $inv_{CD:P:n:mult}$ and $inv_{DS:P:n:mult}$ hold, by assigning the just created elements *a1* and *o1* as next element of elements that do not have successors yet.

2.3 Simultaneous Evolution of Graph Triples

We will now discuss the simultaneous evolution of graph triples by applying the TGG productions of TGG_{CDDS} to an empty graph triple. The resulting graph triple GT_6 (cf. Fig. 3 (a)) is an element of the language of the just introduced TGG. The graph triple is produced by applying production (1) to the empty graph triple and afterwards production (2) to the resulting graph triple. Finally, the following productions are applied: (3), (4), (5), and (5) again. Thus, GT_6 is produced by sequence $SEQ_6 = (p^{(1)}, p^{(2)}, p^{(3)}, p^{(4)}, p^{(5)}, p^{(5)})$.

Production (1) simultaneously creates a class *c1* and a table *t1* and relates them via link *l1*, whereas production (2) creates the subclass *c2* and relates it with the already existing table *t1* via link *l2*. Note that the context in which productions are applied is important (cf. Sect. 5). The context in which production (3) can be applied is either *(c1, l1, t1)* or *(c2, l2, t1)*. In our example, we first choose subclass *c2* as context. Production (3) is applicable in the context of *c2* because neither class *c2* nor table *t1* contain elements at this moment. As a consequence attribute *a3* and column *o3* together with their link *l3* are created. From now on, $p^{(3)}$ is neither applicable in the context of class *c1* nor *c2* due to the NAC in the target domain that blocks because table *t1* already contains column *o3*. In addition, its NAC in the source domain blocks in the context of class *c2* because *c2* contains attribute *a3*. In the next step we apply production (4) in the context of class *c1*. It is applicable because no attribute

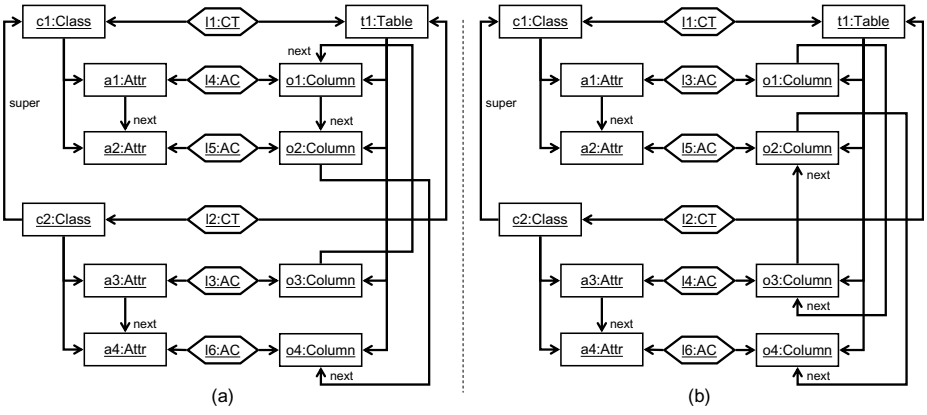


Fig. 3. Schema compliant graph triples GT_6 and GT_6^* produced by TGG_{CDDS}

is currently present in class *c1* and one column *o3* that has no successor yet, is present in table *t1*. So, attribute *a1* and column *o1* are created and related via link *l4* and column *o1* is set as successor of column *o3*. The next two applications of production (5) are again possible in the context of class *c1* and *c2*. We decide to first apply $p^{(5)}$ in the context of *c1* and afterwards in the context of *c2*. This leads to sequence $SEQ_6 = (p^{(1)}@\emptyset, p^{(2)}@c1, p^{(3)}@c2, p^{(4)}@c1, p^{(5)}@c1, p^{(5)}@c2)$ and the final situation depicted in Fig. 3 (a).

The order of columns located in the target domain of GT_6 is *o3*, *o1*, *o2*, and *o4*. This order is determined by the sequence of production applications in a particular context as described above. If the example above is changed such that production (3) is applied in the context of class *c1* and production (4) in the context of class *c2* then $SEQ_6^* = (p^{(1)}, p^{(2)}, p^{(3)}@c1, p^{(4)}@c2, p^{(5)}@c1, p^{(5)}@c2)$. The order of the columns would be *o1*, *o3*, *o2*, and *o4* leading to graph triple GT_6^* (cf. Fig. 3 (b)). We consider these graph triples semantically equivalent according to TGG_{CDDS} because the relative order of attributes in the inheritance structure of classes *c1* and *c2* is not destroyed and the relative order of columns of a relational database does not matter in a "pure" relational calculus.

2.4 Language Translators Based on TGGs

A TGG can be compiled into a pair of *forward and backward graph translators (FGTs/BGTs)*. The generated translators take a graph of the *input domain*, either source or target, and produce a graph triple that consists of the given input graph, the corresponding graph of the *output domain*, either target or source, and the correspondence graph which connects the related source and target graph elements. A translator mainly consists of a set of *graph translation rules* and an algorithm that controls the stepwise translation of a given input graph into the related output graph. Each forward/backward graph translation rule (*FGT/BGT rule*), often called *operational rule*, is directly derived from a

single TGG production[1]. Therefore, TGG productions are split into sets of *local rules* and the aforementioned *translation rules* [1]. Local rules generate graphs of the input domain ensuring that only valid graphs are produced. Hence, local rules are applicable only if NACs are not violated.

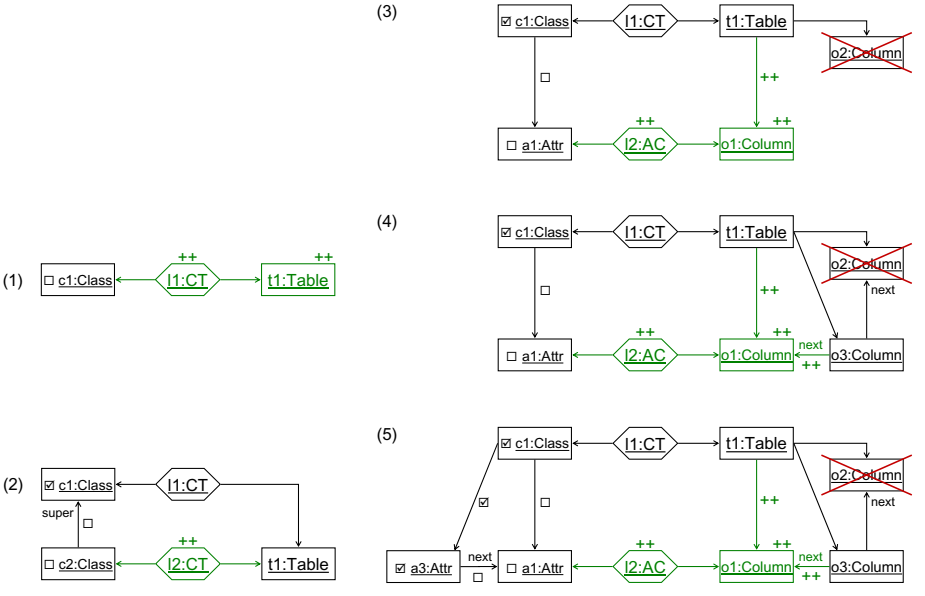


Fig. 4. Forward translation rules derived from TGG_{CDDS}

Figure 4 shows the forward translation rules derived from the productions of TGG_{CDDS}. Translation rules contain elements of source, correspondence, and target domain. The elements of the input domain are readonly as they have been created earlier by a corresponding local rule. Consequently, translation rules only produce elements of the output and correspondence domain. Empty checkboxes denote that the elements next to them are not yet translated, i.e., no corresponding elements in the output domain have been created by another translation beforehand. A translator will mark all elements of the input domain as "translated", right after a translation rule has been applied successfully. Elements that were context elements in a TGG production must be translated before a rule is applicable. This is denoted as enabled checkboxes placed next to or inside these elements. NACs of the input domain may be omitted under certain conditions, as we will learn in Sect. 3.2, whereas NACs of the output domain are retained.

A translation algorithm applies the operational rules to the input graph such that it simulates the simultaneous evolution of the computed graph triple with

[1] For a detailed description of the derivation process we refer to [8].

respect to the given set of TGG productions. Therefore, a translator must be able to determine the order in which elements of the input graph would have been created by a sequence of TGG productions. Guessing the proper choice is one of the difficulties that arise when the simultaneous evolution of graph triples is simulated by a translator. In general, this computation of an appropriate sequence of rules requires a graph grammar parsing algorithm with exponential runtime behavior [9]. Interleaved with the stepwise computation of a sequence of TGG productions and the resulting derivation of the input graph the corresponding sequence of operational rules is executed to generate the related output and correspondence graph instances. For further details concerning a formalization of this process the reader is referred to [1,10]. An example that shows how an FGT translates an input graph is presented in Sect. 5.

2.5 Fundamental Properties of TGGs and Translators

As stated in previous work TGGs should not violate certain design principles in order to be "useful" in practice [5]. Derived translators should be *efficient* and they must be *compatible* with their TGG which in addition must be *expressive* enough. According to [5], efficient translators have polynomial space and time complexity $O\left(m \times n^k\right)$ with m = number of rules, n = size of input graph, and k = maximum number of elements[2] of a rule. This requirement is based on two worst case assumptions: (1) n^k is the worst-case complexity of the pattern matching step of a graph translation rule with k elements. (2) Without starting the pattern matching process for a selected rule we cannot determine whether this rule can be used to translate a just regarded element. As a consequence we require that derived translators do somehow process the elements of an input graph in a given order such that no element has to be regarded and translated more than once. Selecting always somehow the "right" translation rules we do not have to explore multiple translation alternatives using, e.g., a depth-first backtracking algorithm for that purpose. Compatible translators are *consistent* and *complete* with respect to their TGG. Consistency is guaranteed if a translator translates an input graph into a graph triple GT that is always an element of the language $\mathcal{L}(TGG)$ defined by the TGG. Completeness demands that for every graph triple GT that is an element of $\mathcal{L}(TGG)$, a translator is able to produce this graph triple (or an equivalent one) given the graph of the graph triple, which belongs to the translator's input domain. Expressiveness finally requires that the TGG formalism is able to capture all important consistency relationships between studied pairs of graph languages.

As a consequence the original TGG approach is continously extended such that it supports, e.g., handling of attributed typed graphs as well as the definition of productions with NACs. We have learned that NACs are additional preconditions that must be satisfied so a production is applicable. They are, e.g., used to prohibit the construction of graph triples that violate constraints defined in the schema of the source and target domain. But, after more than 15 years

[2] In [5] it is *nodes*. We expand this to *elements* which means nodes and edges.

of TGG research activities we still have problems to handle TGGs with NACs appropriately, i.e., to find the right compromise between expressiveness of TGG productions on the one hand and the introduced consistency, completeness, and efficiency properties of derived translators on the other hand. This contribution introduces, therefore, for the first time a subclass of TGGs with NACs in Sect. 3 that allow for the derivation of efficiently working compatible graph translators. Essentially, the definition and application of TGG productions is restricted in such a way that the here introduced rule application control algorithm (cf. Sect. 5) never has to resolve rule application conflicts by making an arbitrary choice. For this purpose we first replace NACs on the input graph side of a translation by graph constraints, thereby avoiding positive/negative rule application conflicts[3]. Positive/positive rule application conflicts are then eliminated by inspecting the context of those nodes more closely that are just translated by a given rule (cf. Sect. 4)[4]. Inspired by the definition of the double-pushout (DPO) graph grammar approach [11] a new kind of "dangling edge condition" is introduced that blocks the translation of nodes with afterwards still untranslated incident edges under certain conditions.

3 Formalization of Constrained TGGs with NACs

In the preceding section we have informally introduced TGGs with NACs. Furthermore, we already mentioned that [5] does already guarantee consistency but not completeness for the derived translators without introducing a backtracking algorithm, i.e., without trading efficiency for completeness. We will now identify a sort of TGG productions with NACs which do not lead to positive/negative FGT/BGT rule application conflicts for any input graph. This is a first step towards our goal to eliminate all kinds of FGT/BGT rule application conflicts and thereby to guarantee completeness of derived translation functions. For this purpose we extend the TGG formalism as introduced in [1] by NACs that are used to preserve the integrity of graph triples (i.e., resulting graph triples never violate constraints—neither temporary) without destroying the fundamental propositions proved in [1]. Therefore, TGGs will operate on typed constrained graphs and support NACs in a way that derived translators do not violate the mentioned compatibility properties. Permitted NACs will be ignored on the input graph of translation rules assuming that integrity violations of input graphs are captured before a translation process starts.

3.1 Constrained and Typed Graph Grammars with NACs

We start with the basic definitions of constrained, typed graphs on the basis of [11], which are then used for the definition of TGGs that generate triples of typed and constrained graphs in Sect. 3.2.

[3] A positive/negative rule application conflict of two operational rules r and r' w.r.t. specific redexes exists if r creates a graph element that is forbidden by a NAC of r'.

[4] Two operational rules w.r.t. to specific redexes constitute a positive/positive rule application conflict if both rules compete to translate the same graph element.

Definition 1. *Graphs, Graph Morphisms, and Graph Operators.*

A quadruple $G := (V, E, s, t)$ *is a* graph *with elements*$(G) := V \cup E$*, where (1)* V *is a finite set of* nodes *(or* vertices*),* E *is a finite set of* edges*, and (2)* $s, t : E \to V$ *are functions assigning sources and targets to edges. Let* $G := (V, E, s, t), G' := (V', E', s', t')$ *be two graphs. A pair of functions* $h := (h_V, h_E)$ *with* $h_V : V \to V'$ *and* $h_E : E \to E'$ *is a* graph morphism *from* G *to* G'*, i.e.,* $h : G \to G'$*, iff (3)* $\forall e \in E \colon h_V(s(e)) = s'(h_E(e)) \wedge h_V(t(e)) = t'(h_E(e))$

Furthermore, the operators $\subseteq$ *for subgraph,* $\cup$ *for union of graphs with gluing of nodes and edges (with same identifiers), and* $\setminus$ *for the deletion of the removal of graph elements, are defined as usual, and with* $h : G \to G'$ *being a morphism,* $h(G) \subseteq G'$ *denotes that subgraph in* G' *which is the image of* h*.*

Definition 2. *Typed Graph and Type Preserving Graph Morphisms.*

A type graph *is a distinguished graph* $TG := (V_{TG}, E_{TG}, s_{TG}, t_{TG})$*.*

V_{TG} *and* E_{TG} *are called the node and the edge type alphabets, respectively.*

A tuple $(G, type)$ *of a graph* G *together with a graph morphism* $type : G \to TG$ *is called* typed graph*.* G *is called* instance of TG *and* TG *is called* type of G*.*

Given typed graphs G, G' *a typed graph morphism* $g : G \to G'$ *is type preserving iff the diagram shown in Fig. 5 (a) commutes.*

$\mathcal{L}(TG)$ *is the set of all graphs of type* TG*.*

In the following we assume that all graphs with suffix "S", "C", and "T" have type graphs TG_S, TG_C, and TG_T, respectively. Furthermore, we assume that all morphisms between graphs of the same type are type preserving.

Definition 3 introduces constrained graphs. The regarded constraints are typed graph constraints (e.g., OCL invariants) in the spirit of [11], i.e, Boolean formulae over atomic typed graph constraints. A typed graph G fulfills a typed graph constraint c, e.g., if c is evaluated to *true*.

Definition 3. *Constrained Typed Graph.*

A type graph TG *with a set of constraints* $\mathcal{C}$ *defines a subset* $\mathcal{L}(TG, \mathcal{C}) \subseteq \mathcal{L}(TG)$ *of the set of all graphs of type* TG *that fulfill the given set of constraints* $\mathcal{C}$*. The empty graph* $G_\emptyset \in \mathcal{L}(TG, \mathcal{C})$*. Furthermore,* $\overline{\mathcal{L}}(TG, \mathcal{C}) := \mathcal{L}(TG) \setminus \mathcal{L}(TG, \mathcal{C})$ *denotes the set of all graphs of type* TG *that violate a constraint in* $\mathcal{C}$*.*

Based on this definition of constraints we will now define *graph productions with NACs* and graph rewriting. An important property of these productions is that they do not delete any graph elements, i.e, left-hand side L is a subset of right-hand side R. Therefore, they are called monotonic productions.

Definition 4. *Monotonic Graph Productions with NACs.*

The set of all monotonic productions $\mathcal{P}(TG, C)$ *with negative application conditions* $\mathcal{N}$ *for a type graph* TG *with a set of constraints* $\mathcal{C}$ *is defined as follows:* $(L, R, \mathcal{N}) \in \mathcal{P}(TG, \mathcal{C})$ *iff (1)* $L, R \in \mathcal{L}(TG, \mathcal{C}) \wedge L \subseteq R$ *(2)* $\mathcal{N} \subseteq \mathcal{L}(TG) \wedge \forall N \in \mathcal{N} \colon N \supseteq L$

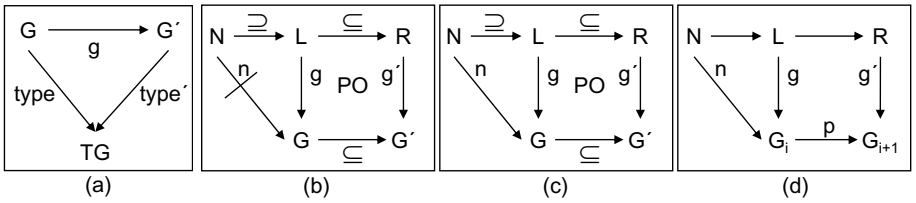


Fig. 5. Diagrams used in Def. 2, Def. 5, Def. 6, and in proof of Corollary 3

Definition 5. *Graph Rewriting for Monotonic Productions with NACs.*
A production $p := (L, R, \mathcal{N}) \in \mathcal{P}(TG, \mathcal{C})$ *rewrites a graph* $G \in \mathcal{L}(TG)$ *into a graph* $G' \in \mathcal{L}(TG)$ *with a redex (match)* $g : L \to G$*, i.e.,* $G \overset{p@g}{\rightsquigarrow} G'$ *iff*
(1) $g' : R \to G'$ *is defined by building the pushout diagram presented in Fig. 5 (b)*
(2) $\neg(\exists N \in \mathcal{N}, n : N \to G : n|_L = g)$*, i.e., there exists no* N *such that mapping* n *is identical to* g *w.r.t. the left-hand side graph* L
(3) all morphisms are type preserving

We will limit productions with NACs to so-called *integrity-preserving productions* in Def. 6 such that NACs are only used to prevent the creation of graphs which violate the set of constraints $\mathcal{C}$. These productions have the important properties that (1) given a valid input graph, a valid output graph is produced, (2) if productions where NACs are eliminated produce a valid graph then the input graph is also valid, and (3) a production that would block due to a NAC otherwise would always produce an invalid graph. Due to contraposition of (1) all invalid output graphs are derived from invalid input graphs. So, integrity-preserving productions produce only invalid output graphs if the input graph was already invalid. Moreover, contraposition of (2) (i.e, (2*)) states that invalid input graphs result in invalid output graphs even if NACs are eliminated from a production; i.e., productions with or without NACs do not repair invalid graphs.

Definition 6. *Integrity-Preserving Productions.*
Let p *be a production* $(L, R, \mathcal{N}) \in \mathcal{P}(TG, \mathcal{C})$ *and* $p^- := (L, R, \emptyset)$ *being the corresponding production of* p *where all negative application conditions have been eliminated. Then,* p *is integrity-preserving iff*
(1) $\forall G, G' \in \mathcal{L}(TG) \wedge G \overset{p}{\rightsquigarrow} G' : G \in \mathcal{L}(TG, \mathcal{C}) \Rightarrow G' \in \mathcal{L}(TG, \mathcal{C})$
(2) $\forall G, G' \in \mathcal{L}(TG) \wedge G \overset{p^-}{\rightsquigarrow} G' : G' \in \mathcal{L}(TG, \mathcal{C}) \Rightarrow G \in \mathcal{L}(TG, \mathcal{C})$
(3) $\forall N \in \mathcal{N}$ *: the existence of the diagram depicted in Fig. 5 (c) with type preserving morphisms* $n|_L = g = g'|_L$ *implies* $G' \in \overline{\mathcal{L}}(TG, \mathcal{C})$

Now, we show for TGG_{CDDS} that the source and target production components satisfy the conditions of Def. 6. We will limit the discussion to the source domain as situations in the target domain are almost identical. All productions satisfy condition (1), i.e., given a valid graph, productions with NACs produce only

valid graphs (discussed already in Sect. 2.2). If productions (3), (4), and (5) without NACs produce valid output graphs under certain conditions then the input graph was also valid due to the fact that none of the productions is able to repair invalid graphs even if their NACs are ignored. Productions (3) and (4) would produce even more (invalid) attributes without successor. Production (5) does not increase the number of attributes without successors that violate multiplicity constraint $inv_{CD:P:n:mult}$, but preserves the number of attributes that violate $inv_{CD:P:n:mult}$. Therefore, condition (2) is satisfied. Condition (3) of Def. 6 is satisfied because a blocking NAC of productions (3) and (4) prevents the production of an additional attribute without successor. Therefore, violation of $inv_{CD:C}$ is prevented. In addition, a blocking NAC in production (5) prevents violation of the multiplicity constraint of a "Precedes" edge $inv_{CD:P:n:mult}$. Therefore, the set of productions of TGG_{CDDS} is *integrity-preserving*.

Finally, Def. 7 states how graph grammars produce constrained typed graphs.

Definition 7. *Language of Typed and Constrained Graph Grammars.*

A graph grammar $GG := (TG, \mathcal{C}, \mathcal{P})$ *over a type graph* TG*, a set of constraints* $\mathcal{C}$*, and a finite set of integrity-preserving productions* $\mathcal{P} \subseteq \mathcal{P}(TG, \mathcal{C})$*, with* $G_\emptyset$ *being the empty graph, generates the following language of graphs*
$\mathcal{L}(GG) := \{G \in \mathcal{L}(TG, \mathcal{C}) | G_\emptyset \overset{p_1}{\rightsquigarrow} G_1 \overset{p_2}{\rightsquigarrow} ... \overset{p_n}{\rightsquigarrow} G_n = G \text{ with } p_1, ..., p_n \in \mathcal{P}\}$

The language that is generated by a graph grammar GG as defined in Def. 7 (i.e., the graphs that are producible by the grammar) is a subset of the set of graphs of type TG that fulfill the given set of constraints $\mathcal{C}$.

Corollary 1. $\mathcal{L}(GG) \subseteq \mathcal{L}(TG, \mathcal{C})$

Proof. Follows from Def. 6 and directly from Def. 7. □

Furthermore, the language $\mathcal{L}(GG^-)$ generated by a graph grammar where NACs of productions have been eliminated contains at least the same graphs as the language $\mathcal{L}(GG)$ generated by this graph grammar with NACs.

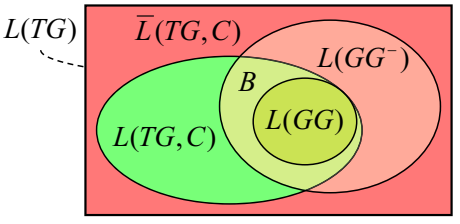
Corollary 2. *Let* GG^- *be a graph grammar derived from a graph grammar* GG*, where all negative application conditions of productions have been eliminated. Then* $\mathcal{L}(GG^-) \supseteq \mathcal{L}(GG)$.

Proof. Due to the fact that a valid application of a production p with NACs is also a valid application of the production p^- where NACs are ignored, $\mathcal{L}(GG^-)$ is at least as large as $\mathcal{L}(GG)$. □

Moreover, $\mathcal{L}(GG)$ is the intersection of the graphs producible by $\mathcal{L}(GG^-)$ and the set of graphs of type TG that fulfill the given set of constraints $\mathcal{C}$.

Corollary 3. *Let* GG^- *be a graph grammar derived from a graph grammar* GG *as defined in Def. 7, where all negative application conditions of productions have been eliminated. Then* $\mathcal{L}(GG) = \mathcal{L}(GG^-) \cap \mathcal{L}(TG, \mathcal{C})$.

Proof. Due to Corollary 1 and Corollary 2 the intersection of sets of graphs defined by $\mathcal{L}(TG,\mathcal{C})$, $\overline{\mathcal{L}}(TG,\mathcal{C})$, $\mathcal{L}(GG)$, and $\mathcal{L}(GG^-)$ looks like depicted:



Therefore, we only have to show that $\mathcal{B} := \mathcal{L}(GG^-) \cap \mathcal{L}(TG,\mathcal{C}) \setminus \mathcal{L}(GG)$ is empty. Let $G \in \mathcal{B}$, i.e., G is generated by a sequence of production applications

$$G_\emptyset \rightsquigarrow \ldots \rightsquigarrow G_i \overset{p^-}{\rightsquigarrow} G_{i+1} \rightsquigarrow \ldots \rightsquigarrow G$$

with $p = (L, R, \mathcal{N})$ being a production of GG and $p^- = (L, R, \emptyset)$ being the corresponding production of GG^- such that $\exists N \in \mathcal{N}$ so the diagram shown in Fig. 5 (d) commutes, i.e., p is blocked by N, but p^- rewrites G_i into G_{i+1}.
$\Rightarrow G_{i+1} \in \overline{\mathcal{L}}(TG,\mathcal{C})$. This is a direct consequence of Def. 6 (3), which requires that the application of p^- produces a graph G_{i+1}, which violates at least one constraint if the application of p is blocked by its NAC N.
$\Rightarrow G \in \overline{\mathcal{L}}(TG,\mathcal{C})$. This is a direct consequence of Def. 6 (2^*) because all graphs on the derivation path from G_{i+1} to G (including G) are invalid due to the fact that productions of GG^- preserve the property of a graph to violate some constraint. This leads to contradiction. □

As a consequence of Def. 6 and due to Corollary 3, we can either check NACs during the execution of a (TGG) production to prohibit the violation of graph constraints immediately or check potentially violated graph constraints after a sequence of graph rewriting steps that simply ignore NACs; for a more detailed discussion of the relationship of (positive) pre- and postconditions of graph transformation rules and graph constraints we refer to [12].

3.2 Constrained and Typed Triple Graph Grammars with NACs

Having introduced definitions and properties of graph grammars with NACs for languages of typed constrained graphs we now present the corresponding definitions of TGGs with NACs for typed constrained graph triples. For this purpose we have to replace the definitions of simple graphs and graph grammars in [1] by the more elaborate definitions given in Sect 3.1.

Definition 8 describes the conditions typed graph triples must satisfy. A graph triple consists of three graphs. Each graph is in the set of graphs of a particular language, i.e., conforms to a graph schema defined by a certain type graph. In addition, two morphisms h_S and h_T relate elements of the correspondence graph with elements of the source and target graph. Constraints for correspondence graphs are disregarded in Def. 8, but can be added easily if needed.

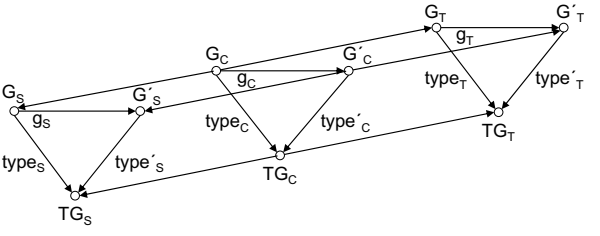
Definition 8. *Constrained Typed Graph Triple.*

Let $\mathcal{L}(TG_S, \mathcal{C}_S)$ and $\mathcal{L}(TG_T, \mathcal{C}_T)$ be languages of source and target graphs with constraints, whereas $\mathcal{L}(TG_C)$ defines a language of correspondence graphs that relate pairs of source and target graphs.

$GT := (G_S \overset{h_S}{\leftarrow} G_C \overset{h_T}{\rightarrow} G_T) \in \mathcal{L}(TGG)$ is a properly typed graph triple *iff*
(1) $G_S \in \mathcal{L}(TG_S, \mathcal{C}_S)$, (2) $G_C \in \mathcal{L}(TG_C)$, (3) $G_T \in \mathcal{L}(TG_T, \mathcal{C}_T)$
(4) $h_S : G_C \rightarrow G_S$, (5) $h_T : G_C \rightarrow G_T$

Definition 9. *Type Preserving Graph Triple Morphisms.*

A type graph triple *$TGT := (TG_S \overset{h_S}{\leftarrow} TG_C \overset{h_T}{\rightarrow} TG_T)$ is a distinguished graph triple. TGT together with morphisms $type_S : G_S \rightarrow TG_S$, $type_C : G_C \rightarrow TG_C$, $type_T : G_T \rightarrow TG_T$ is called type of GT. A graph triple morphism (g_S, g_C, g_T) with $g_S : G_S \rightarrow G'_S$, $g_C : G_C \rightarrow G'_C$, $g_T : G_T \rightarrow G'_T$ is type preserving iff the so-called "toblerone" diagram*



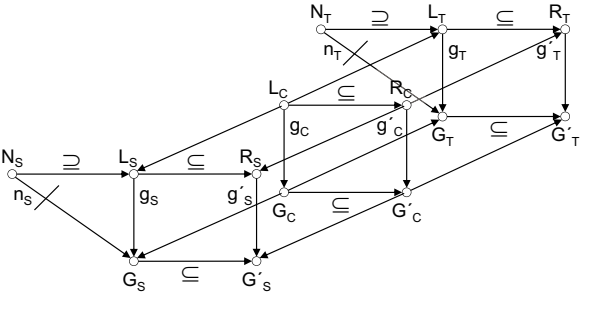
commutes. $\mathcal{L}(TGT)$ is the set of all graphs of type TGT.

Now, we are ready to lift graph rewriting (cf. Def. 5) based on monotonic productions (cf. Def. 4) and integrity-preserving productions (cf. Def. 6) to graph triple rewriting in Def. 10.

Definition 10. *Integrity-Preserving Graph Triple Rewriting.*

Let $p := (p_S \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$ be a production triple with NACs *and*

(1) $p_S := (L_S, R_S, \mathcal{N}_S) \in \mathcal{P}(TG_S, \mathcal{C}_S)$ be an integrity-preserving production
(2) $p_C := (L_C, R_C, \emptyset) \in \mathcal{P}(TG_C, \emptyset)$ be a simple production
(3) $p_T := (L_T, R_T, \mathcal{N}_T) \in \mathcal{P}(TG_T, \mathcal{C}_T)$ be an integrity-preserving production
(4) $h_S : R_C \rightarrow R_S$, $h_S|_{L_C} : L_C \rightarrow L_S$ and (5) $h_T : R_C \rightarrow R_T$, $h_T|_{L_C} : L_C \rightarrow L_T$
The application of such a production triple to a graph triple GT produces another graph triple GT', i.e., $GT \overset{p}{\rightsquigarrow} GT'$, which is uniquely defined (up to isomorphism) by the existence of the following "pair of cubes" diagram:



This diagram consists of commuting square-like subdiagrams only and contains a pushout subdiagram for each application of a production component *(i.e,* p_S*,* p_C*, and* p_T*) to its corresponding* graph component*.*

For the details of the definition and the proof that production triples applied to graph triples at a given redex always produce another graph triple uniquely defined up to isomorphism, cf. [1]. NACs introduced here do not destroy the constructions and proofs introduced in [1] due to the fact that they do not (further) influence the application of a production to a given graph (triple) after all NAC applicability checks have been executed. Based on the presented definitions we introduce *typed triple graph grammars* and their languages. For reasons of readability we omit the prefix "typed" throughout the rest of this contribution.

Definition 11. *Triple Graph Grammar and Triple Graph Grammar Language. A* triple graph grammar *TGG over a triple of type graphs* (TG_S, TG_C, TG_T) *is a tuple* $(P, GT_\emptyset)$*, where* P *is the set of its TGG productions and* $GT_\emptyset$ *is the empty graph triple. The language* $\mathcal{L}(TGG)$ *is the set of all graph triples that can be derived from* $GT_\emptyset := (G_\emptyset \xleftarrow{\varepsilon} G_\emptyset \xrightarrow{\varepsilon} G_\emptyset)$ *using a finite number of TGG production rewriting steps.*

We can now show that a triple graph grammar TGG^-, where all NACs (that prevent the creation of graph triples that violate graph constraints) are removed from TGG productions, produces the same set of constrained graph triples that is produced by the unmodified triple graph grammar TGG.

Theorem 1. *With* $\mathcal{L}(TGG)$ *being the language of graph triples generated by a triple graph grammar TGG over* (TG_S, TG_C, TG_T) *we can show:*
(1) for all $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$*:*
$G_S \in \mathcal{L}(TG_S, \mathcal{C}_S)$*,* $G_C \in \mathcal{L}(TG_C)$*,* $G_T \in \mathcal{L}(TG_T, \mathcal{C}_T)$
(2) with TGG^- *being the triple graph grammar derived from TGG where all NACs of productions have been removed:*
$(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG) \Leftrightarrow (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG^-) \wedge$
$(G_S, G_C, G_T) \in \mathcal{L}(TG_S, \mathcal{C}_S) \times \mathcal{L}(TG_C) \times \mathcal{L}(TG_T, \mathcal{C}_T)$

Proof. Follows from Def. 10 (which lifts graph to graph triple rewriting) and Corollaries 1 and 2. The proof is analogous to the proof of Corollary 3. □

It is a direct consequence of Theorem 1 that checking of NACs can be replaced by checking integrity of generated graphs with respect to their sets of constraints and vice versa. This observation directly affects translators derived from a given TGG as follows: According to [1], a production triple p may be split into pairs of production triples (r_I, r_{IO}), where r_I is an *(input-) local rule* and r_{IO} its corresponding *(input-to-output domain) translation rule*, with $GT \overset{p}{\rightsquigarrow} GT' \Leftrightarrow GT \overset{r_I}{\rightsquigarrow} GT_I \overset{r_{IO}}{\rightsquigarrow} GT'$. Forward translation is based on (r_S, r_{ST}), whereas (r_T, r_{TS}) is used in the reverse direction. To rewrite the source graph only, the *source-local production triple*, i.e., *source-local rule* $r_S := (p_S \xleftarrow{\varepsilon} (\emptyset, \emptyset, \emptyset) \xrightarrow{\varepsilon} (\emptyset, \emptyset, \emptyset))$ is applied. The *source-to-target domain translating production triple*, i.e., *forward*

graph translation rule r_{ST} keeps the source graph unmodified but adjusts the correspondence and target graph as follows: the effect of applying first r_S and then r_{ST} to a given graph triple is the same as applying p itself if (and only if) we keep the source domain redex, i.e., the morphism g'_S, fixed. Thanks to Theorem 1 the source component of r_{ST} does not have to check any NACs on the source graph as long as any regarded source graph does not violate any graph constraints, i.e., as long as it has been constructed by means of integrity-preserving productions only. As a consequence, we need no longer care about positive/negative rule application conflicts on the source side when translating a source graph into a related target graph.

Definition 12. *Forward Graph Translation Rules.*

With p being constructed as listed above in Def. 11 the derived forward graph translation rule (FGT rule) *is* $r_{ST} := (p^-_{S,id} \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$ *with components:*

(1) $p^-_{S,id} := (R_S, R_S, \emptyset)$*, i.e., the source component* p_S *of* p *without any NACs that matches and preserves the required subgraph of the source graph only*

(2) $p_C := (L_C, R_C, \emptyset)$*, i.e., the unmodified correspondence component of* p

(3) $p_T := (L_T, R_T, \mathcal{N}_T)$*, i.e., the unmodified target component of* p

For a detailed definition of r_{ST} that includes the morphisms between its rule components as well as for the definition of r_S/r_T and the definition of a *backward graph translation rule* (BGT rule) r_{TS} the reader is referred to [1]. The definitions presented there can be adapted easily to the scenario of integrity-preserving graph triple rewriting as done here for the case of FGT rules r_{ST}.

Definition 13 introduces the so-called *local completeness criterion* of the source domain which must be satisfied by the productions of a TGG. Essentially the definition requires that any sequence $SEQ^n_{i=1}(r_{S,i})$ can be completed to a sequence $SEQ^n_{i=1}(r_{ST,i})$ of derivation steps of a graph triple GT that exactly mimics the derivation of its source graph G_S. This criterion will be used later on in Sect. 5 to prove the completeness of the introduced algorithm that translates a given source graph G_S into a compatible target graph G_T together with a graph G_C that connects G_S and G_T appropriately.

A similar criterion of the target domain can be defined accordingly. The productions of TGG_{CDDS} satisfy both source and target criterions.

Definition 13. *Source-Local Completeness Criterion.*

A triple graph grammar TGG fulfills the source-local completeness criterion *iff for all* $GT_i := (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ *and* $p := (p_S \leftarrow p_C \rightarrow p_T) \in P$ *with* $G_S \overset{p_S @ g_S}{\rightsquigarrow} G'_S$ *exists* $p^* := (p^*_S \leftarrow p^*_C \rightarrow p^*_T) \in P$*,* $g^* := (g^*_S, g^*_C, g^*_T)$*, and* $GT^*_{i+1} := (G'_S \leftarrow G'_C \rightarrow G'_T) \in \mathcal{L}(TGG)$ *such that* $GT_i \overset{p^* @ g^*}{\rightsquigarrow} GT^*_{i+1}$.

The local completeness criterions demand that for each local graph (G_S or G_T) of all graph triples $GT \in \mathcal{L}(TGG)$, which is rewritten by the local component of a production p, there must be at least one production p^* (p^* may equal p) which rewrites GT. Therefore, each match $g'_I(R_I \backslash L_I)$ of an input component $p^-_{I,id}$ of a translation rule r_{IO} that identifies not yet translated elements in an

input graph can be completed to a full match on the correspondence and output graphs. This is due to the fact that at least one local rule r_I (derived from a production p) exists that has created the matched yet untranslated elements in the input graph. According to the local completeness criterion a production p^* exists from which a local rule r_I^* is derived that creates the same elements as r_I. Hence, a translation rule r_{IO}^* exists that has an equivalent input component to r_{IO} which is able to translate the matched not yet translated elements. As a consequence, derived translation rules are complete, i.e., they can be used to translate any given input graph of a TGG language into a properly related graph. Furthermore, Theorem 1 guarantees the consistency of derived translation rules even if NACs are omitted. Consequently, derived rules never translate input graphs of a TGG language into output graphs such that the resulting graph triple is not an element of the just regarded TGG language.

Due to these achievements we are able to build translators that are consistent and complete with respect to their TGG. During the translation process a translator parses a given input graph in order to find a valid sequence of translation rules that mimics the derivation of the input graph. Although the TGG productions contain NACs these can be safely ignored in the parsing process in the case of integrity-preserving productions. Therefore, positive/negative rule application conflicts are prevented on the input graph. Positive/negative conflicts on the output graph will not lead to dead-ends (i.e., wrong translation alternatives which require backtracking) during parsing because the local completeness criterion guarantees that for each remaining untranslated element in the input graph, created by a local rule, a translation rule exists that is able to translate these elements. Unfortunately, we still have to solve one problem: in general we are only able to guarantee the completeness of a derived graph translator if we explore an exponential number of derivation paths (w.r.t. the size of a given input graph) due to the remaining positive/positive rule application conflicts. The following section will solve this efficiency problem for a sufficiently large class of TGGs (from a practical point of view) by introducing a new application condition for translation rules. This condition rules out any situation, where more than one rule can be used to translate a just regarded node of the input domain in a related subgraph of the output domain.

4 Dangling Edge Condition (DEC)

Translators derived from a TGG face certain difficulties concerning the selection of an appropriate sequence of translation rules in the presence of positive/positive rule application conflicts. Reconsider our triple graph grammar TGG_{CDDS} from Sect. 2. An FGT derived from TGG_{CDDS} translates class diagrams to database schemata. Figure 6 (a) depicts a graph that consists of two classes *c1* and *c2*. The empty checkboxes[5] denote that the elements next to them are not yet translated. The graph is valid, as it is derivable by applying productions (1) and afterwards (2) of TGG_{CDDS} to the empty graph triple. The graph is given

[5] Fig. 6 uses an alternative representation of (not) translated nodes.

as input graph to the FGT. First, the translator applies the FGT rule derived from production (1), which translates class *c1*. Next, both rules (1) and (2) are applicable in the context of class *c2*. If the translator chooses to translate class *c2* via rule (1) the source graph would contain two translated classes with an untranslated edge between them (cf. Fig. 6 (b)). Unfortunately, no rule exists that is able to translate the remaining untranslated edge *e1*. So, the translator produced a so-called *dangling edge* in the source graph. Consequently, the translator states at the end of the translation process that it is not able to translate the (valid) input graph completely due to this *dangling edge*.

Fig. 6. (a) Input graph given to the FGT, (b) Input graph with two translated nodes

Whenever constellations in the input graph appear, where two or more rules are applicable that translate overlapping sets of input graph elements, translation algorithms are demanding for help to select the appropriate rule. We propose an extension that is inspired by building parsers for compilers and related techniques for parsing words that are passed to the compiler. Typically, top-down and bottom-up parsers decide on more information than just the recent input: they take a *look-ahead* into account. In the following subsection we introduce a so-called *dangling edge condition* (DEC) that prevents the application of a rule if the rule would produce a dangling edge. TGG translators produce dangling edges if an edge is still untranslated at the end of the translation process. So, translators must ensure that before applying a rule another translation rule exists that is able to translate this currently "dangling" edge later on. This DEC is inspired by an analogous condition in DPO approaches, which explicitly prohibits deleting a node without deleting all incident context edges as part of the same rule application step. This way, our DEC eliminates positive/positive rule application conflicts. We restrict our focus to forward translators in the sequel, but all concepts and ideas can be transferred to backward translators as well.

The core idea of the DEC is that several productions may be applicable such that their matches overlap in some node. If the production with the smaller match is applied, incident edges cannot be translated later on. The DEC resolves conflicts where context-sensitive productions create one primary node that is connected via new edges to at least one context node. It does not offer a solution for those cases where the created nodes are not connected. In the following, we regard TGG productions that create only one primary node on each side and do not contain additional secondary elements[6]. Primary nodes of context-sensitive productions must be connected to at least one context node. The graphs

[6] Allowing additional secondary elements would require in depth discussions and special handling in Algorithm 1 which we had to omit due to lack of space.

that result by applying such productions are either graph structures that are not connected to other structures (in case of applying initial context-free productions like production (1) of TGG_{CDDS}) or connected graph structures (in case of applying context-sensitive productions (2) to (5) of TGG_{CDDS}).

4.1 Formal introduction to LNCC and DEC

As shown at the beginning of Sect. 4, application of certain translation rules may lead to invalid graph triples since some edges in the graph of the input domain remain untranslated. Based on this observation we define for the source graph of a TGG the so-called *Legal Node Creation Context* relation with a look-ahead of one $LNCC_S(1)$[7] that will be used to control the selection and application of FGT rules. A relation $LNCC_T(1)$ used by BGTs is constructed similarily. TGG productions can be broken down to certain fragments, where at most two nodes make up a part of the production. Elements of $LNCC_S(1)$ are *4-tuples* that represent certain kinds of source graph production fragments. The first and third component of a tuple represent the type of the node that is the source and target of an edge e created by a production, respectively. The type of this edge e is used as second component. The fourth component denotes whether the source node, target node, or both nodes are used as context in the production fragment. Tuples of $LNCC_S(1)$ are derived from a given TGG as follows:

Definition 14. *Legal Node Creation Context with a look-ahead of 1.*
$LNCC_S(1) \subseteq V_{TG_S} \times E_{TG_S} \times V_{TG_S} \times \{s, t, st\}$ *is the smallest* legal node creation context relation *for the source graph of a given TGG such that* $(vt_s, et, vt_t, c) \in LNCC_S(1)$ *iff*

(1) $\exists$ *TGG production* $((L_S, R_S, \mathcal{N}_S) \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$ *that creates edge* $e \in R_S \backslash L_S$ *with at least one incident already existing context node* $s(e)$ *or* $t(e) \in L_S$
(2) $vt_s = type(s(e))$, *(3)* $et = type(e)$, *(4)* $vt_t = type(t(e))$
(5) $c \in \{s, t, st\}$, *with the following semantics:*
(5.1) s: $s(e) \in L_S$, $t(e) \in R_S \backslash L_S$
(5.2) t: $t(e) \in L_S$, $s(e) \in R_S \backslash L_S$ *(5.3) st:* $s(e), t(e) \in L_S$

Figures 7 (a), (b), and (c) identify all possible node and edge constellations that contribute tuples to $LNCC_S(1)$. In addition, Figs. 7 (d), (e), and (f) depict those production fragments that do not contribute any tuples to $LNCC_S(1)$.

The motivation behind the definition of $LNCC_S(1)$ is to block a translation of a node of the source graph that has incident edges that are not translated in the same step and that cannot be translated later on (i.e., to avoid dangling edges). This situation occurs if a TGG contains overlapping productions (e.g., productions $p^{(1)}$ and $p^{(2)}$ of TGG_{CDDS}). These productions are applicable in the same context and create a node of the same type (both $p^{(1)}$ and $p^{(2)}$ create nodes of type "Class") but at least one production creates an edge that relates the new

[7] We plan to introduce $LNCC_S(n)$ with a look-ahead of $n > 1$ that also takes indirectly referenced nodes into account in future work.

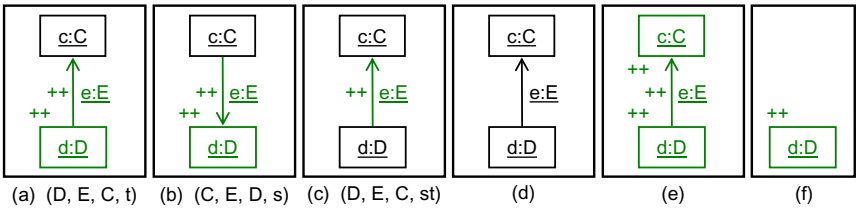


Fig. 7. TGG production fragments relevant and irrelevant for $LNCC_S(1)$

node to an already existing node ($p^{(2)}$ creates an edge from the new subclass to its superclass). Therefore, a translator that applies one of the rules derived from these productions would destroy the match of the other rule and potentially leave an untranslatable edge. In order to identify such dangling edge situations, TGG production fragments must be inspected which create edges where the source or the target of the edge already exists, i.e., is used as context (cf. Figs. 7 (a), (b), and (c)). Translation rules derived from TGG productions containing these fragments have the potential to translate edges of the input graph using one or two already translated incident nodes as context. As patterns (d) to (f) do not translate such edges they can be neglected. Pattern (a) depicts a production fragment in which node c is the already existing context for the new node d and d is the source of the new edge e ($s(e) = d$). In production fragment (b) the direction of the edge is changed: $s(e) = c$. Pattern (c) depicts a situation where a new edge between nodes c and d is created, i.e., both nodes are used as context.

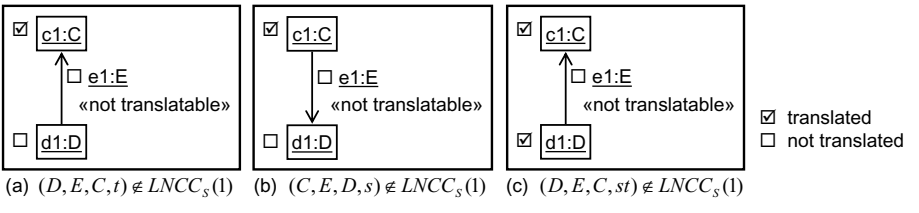


Fig. 8. Patterns in input graph that violate DEC(1)

Whenever we encounter a not translated edge with an already translated incident node, we will use the relation $LNCC_S(1)$ to check whether an FGT rule exists that can be used later on to translate the regarded edge. If $LNCC_S(1)$ does not contain an appropriate tuple then the just regarded edge cannot be translated. On the other hand, the existence of an appropriate tuple does not guarantee that the edge is translatable. This is due to the fact that FGT rules r_{ST} containing a tuple are only applicable if a match of a rule's complete left-hand side ($R_S \leftarrow L_C \rightarrow L_T$) is found in the host graph triple and no NAC in the target domain blocks. In general we have to restrict the application of translation rules such that the situations depicted in Fig. 8 are avoided:

(a) Node $d1$ is not translated yet but $c1$ (the target of $e1$) is and there exists no rule with production fragment (D, E, C, t) that may translate $e1$ later on.
(b) Node $d1$ is not translated yet but $c1$ (the source of $e1$) is and there exists no rule with production fragment (C, E, D, s) that may translate $e1$ later on.
(c) Nodes $d1$ (source) and $c1$ (target) are both translated and there exists no rule with production fragment (D, E, C, st) that may translate $e1$ later on.

Therefore, the application of a translation rule must satisfy certain application conditions given in Def. 15 including the *Dangling Edge Condition* ($DEC(1)$).

Definition 15. *Rule application conditions for FGTs with a look-ahead of 1.*
Let TX be the set of already translated elements of the source graph G_S, $e \in E_S$, and p be a TGG production $((L_S, R_S, \mathcal{N}_S) \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$. Thus, for each match g'_S of translation rule r_{ST} in G_S the rule application conditions (1) to (3) must hold including the dangling edge condition *$DEC(1)$ that consists of the subconditions $DEC_1(1)$, $DEC_2(1)$, and $DEC_3(1)$ in order to apply r_{ST} to $(G_S \leftarrow \ldots \rightarrow \ldots)$:*

(1) $g'_S(L_S) \subseteq TX$ *(context elements are already translated)*
(2) $\forall x \in g'_S(R_S \backslash L_S) : x \notin TX$ *(no element x shall be translated twice)*
(3) $TX' := TX \cup g'_S(R_S \backslash L_S)$ *(TX is extended with translated elements)*
($DEC_1(1)$) $\forall e \notin TX'$ *where* $s(e) \in TX', t(e) \notin TX'$:
$(type(s(e)), type(e), type(t(e)), s) \in LNCC_S(1)$
($DEC_2(1)$) $\forall e \notin TX'$ *where* $t(e) \in TX', s(e) \notin TX'$:
$(type(s(e)), type(e), type(t(e)), t) \in LNCC_S(1)$
($DEC_3(1)$) $\forall e \notin TX'$ *where* $s(e), t(e) \in TX'$:
$(type(s(e)), type(e), type(t(e)), st) \in LNCC_S(1)$

Def. 15 thus introduces a rather straightforward way to decide if a translation rule shall be applied or not just by looking at the 1-context of a to-be-translated node. By adding this condition to the translation algorithm defined in [5] (cf. Algorithm 1 in Sect. 5), we are able to reduce the number of situations significantly, where we were forced to choose one of the applicable rules nondeterministically and run into dead-ends due to the wrong choice. In general, Def. 15 is not able to resolve all positive/positive conflicts, i.e., there may be multiple rules that are able to translate a node using different matches, i.e., matches containing different to-be-translated elements. Therefore, Algorithm 1 will abort in this case. Alternatively, the user could be asked which of these elements should be translated or rule priorities [8] can be used to reduce the number of different matches if more than one rule is applicable by filtering matches of rules with low priority.

Though, the algorithm permits multiple *locally-applicable rules*, i.e., rules that translate the same elements. A locally-applicable rule is either applicable also on the whole graph triple or its application is prevented, e.g., due to NACs in the output component. The set of productions of TGG_{CDDS} contains multiple locally-applicable rules. FGT rules (3) and (4) are both applicable in the context of the first attribute of a class. Likewise, BGT rules (4) and (5) are both applicable in the context of the non-first column of a table. These rules are *disjoint applicable*, i.e., only one of the locally-applicable rules is applicable on the whole graph triple (cf. forward translation example in Sect. 5). In general, multiple

locally-applicable rules need not to be disjoint applicable because they translate the same elements. Executing one of the locally-applicable rules nondeterministically does not lead into dead-ends due to the local completeness criterion and the same reason why positive/negative conflicts on the target side do not lead into dead-ends (cf. Def. 13 and subsequent discussion).

4.2 Dangling Edge Condition by Example

Now, we show by example that checking for dangling edges helps deciding which rule should be applied by translators derived from a TGG if multiple rules are applicable at overlapping matches. Therefore, we consider again the FGT derived from TGG_{CDDS} and the input graph depicted in Fig. 6 (a) already discussed at the beginning of Sect. 4. As we have already shown, both translation rules (1) and (2) are applicable after applying rule (1) to this input graph. Based on the classification scheme of Fig. 7 and Def. 14 we construct the set of tuples from the TGG productions of TGG_{CDDS} which results in $LNCC_S(1) = \{(Class, Inherits, Class, t), (Class, Contains, Attr, s), (Attr, Precedes, Attr, s)\}$. Next, we pretend to apply rule (1) in the context of class *c2*. Then, we calculate the set $inc(c2) = \{e1\}$ which contains incident edges of *c2* that are not yet translated. We must check whether all edges in $inc(c2)$ are translatable by further rewriting steps, i.e., whether $DEC(1)$ is satisfied. As both source and target of *e1* are already translated, a tuple must exist in $LNCC_S(1)$ that satisfies subcondition $DEC_3(1)$. Therefore, the tuple $(Class, Inherits, Class, st)$ must be in $LNCC_S(1)$ which is not the case. As a consequence, we do not apply FGT rule (1), because this would result in a dangling edge (cf. Fig. 6 (b)) and proceed pretending to apply rule (2). In this case $inc(c2) = \emptyset$. So, the rule application conditions given in Def. 15 are satisfied, i.e., there are no dangling edges. Concluding, we were able to translate the input graph completely due to the fact that the DEC prohibited selecting a wrong translation rule match.

5 Extended Translation Algorithm with DEC

In this section we extend the algorithm presented in [5] so it handles NACs as presented in Sect. 3 and checks the dangling edge condition (cf. Sect. 4). We discuss the extended algorithm (cf. Algorithm 1) by translating the graph triple shown in Fig. 10 (a) with the FGT derived from TGG_{CDDS}. Each translator implements procedure $evolve : GT_{in} \rightsquigarrow^* GT_{out}$ which simulates the simultaneous evolution of a given graph triple GT_{in}. The input graph triple GT_{in} is either $(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset)$ in case of an FGT or $(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T)$ in case of a BGT[8]. The input graph G_{input} is either G_S or G_T depending on the type of translator (FGT/BGT), whereas the output graph G_{output} is either G_T or G_S. Procedure *evolve* assumes that the underlying TGG is integrity-preserving and that the input graph GT_{in} was produced by a sequence of input-local rules r_I,

[8] In the general case, when incremental updates are performed with a translator, the correspondence graph and the graph of the opposite domain need not to be empty.

i.e., $G_{input} \in \mathcal{L}(GG_I)$. *Evolve* is able to cope with situations where the underlying TGG or the input is invalid. It throws errors if it detects an invalid TGG specification and exceptions in case of invalid inputs. A valid translation produces an output graph triple $GT_{out} = (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$. Therefore, *evolve* calls subroutine *translate(GraphTriple)* which in turn calls procedure *translate(Node)* for all nodes in the input graph G_{input}. Resulting graph triples of invalid translations are undefined. Algorithm 1 uses so-called *core rules* (cf. [5]) to determine matches of translation rules in the input graph. A core rule is closely related to the input component $p^-_{I,id}$ (either $p^-_{S,id}$ or $p^-_{T,id}$) of a translation rule (cf. Def. 12). Fig. 9 shows the core rules derived from TGG_{CDDS} which are used by the forward translator. A core rule looks up the context elements of a given primary element in G_{input}, which may or may not be translated already but must be translated so the primary element is translatable (cf. Def. 15 (1)). Core rules only contain elements of the input graph. NACs are not contained in a core rule. The primary element and additional incident edges must not be translated yet. This is indicated by the empty checkboxes next to these elements.

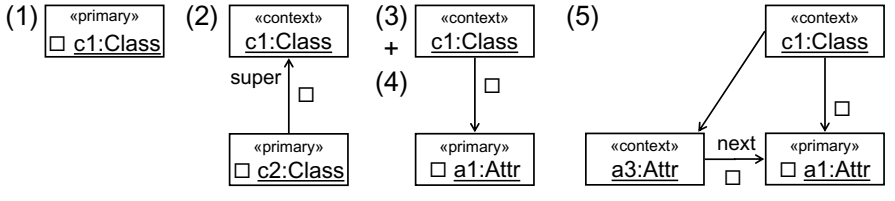


Fig. 9. Core rules of the FGT derived from TGG_{CDDS}

The algorithm starts translating G_{input} (cf. Fig. 10 (a)) with node *c1*. It may start with any other node as well because it recursively translates the context of the current node before it translates the node itself. First, the algorithm determines *appropriate rules* from the set of *candidate rules*. A rule is a candidate if its primary node in the input domain is type compatible with the to-be-translated primary node. An appropriate rule has at least one *core match* which contains the primary node. A core match satisfies Def. 15 (2), i.e., all to-be-translated elements are not translated yet. If multiple matches of one rule in the input graph exist[9], the algorithm checks for every match if the rule is appropriate. In order to be appropriate, every context node required by the primary node is recursively translated. But, the context is only translated if the dangling edge condition would be satisfied afterwards (cf. Def. 15 DEC(1)). Next, the algorithm determines *applicable rules* from the set of appropriate rules. The application condition Def. 15 DEC(1) has to be reassured because it might have been invalidated due to potential competing recursive context translations. In addition, the core match of an applicable rule must be completed in the rule's left-hand side

[9] In TGG_{CDDS} at most one core match exists for any rule in a valid input graph.

(i.e., input, link, and output domain) and the NACs in the output domain must not block. If no applicable rule is determined then either a match exists in the input domain but it may not be completed or the to-be-translated node is not even locally translatable. In the first case the set of TGG productions violates the local completeness criterion (cf. Def. 13), in the latter case the input graph is invalid. If multiple rules are applicable at some completed match, the algorithm ensures that their to-be-translated elements in the core match are identical. Otherwise it aborts with an error as this might lead into dead-ends (cf. Sect. 4.1). It is up to the developer of a set of TGG productions to guarantee that this will never happen in practice. Finally, the algorithm translates the primary node. It selects one entry from the set of applicable rules, applies the rule at its match, and extends the set of translated elements (cf. Def. 15 (3)).

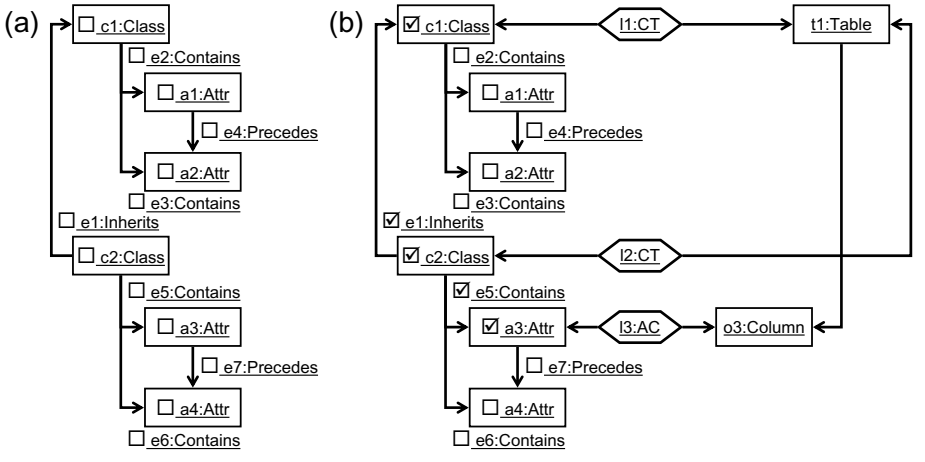


Fig. 10. Snapshots of the translation process during forward translation

FGT rule (1) (i.e., $r^{(1)}_{ST}$) is the only candidate that has the capability to translate *c1* because no match exists for the other candidate rule (2). The algorithm checks whether application of rule (1) satisfies DEC(1). Therefore, it determines the elements in G_{input} created by the corresponding local rule $r^{(1)}_{S}$ (i.e., $\{c1\}$) and joins these elements, the required context elements (i.e., $\emptyset$), and the set of currently translated elements (i.e., $\emptyset$) which results in set $TX' := \{c1\}$. Then, it checks whether condition DEC(1) is satisfied for all not translated incident edges of node *c1*, i.e., $inc(c1) := \{e1, e2, e3\}$. According to Def. 15, the required tuples for $inc(c1)$ are $(Class, Inherits, Class, t)$ and $(Class, Contains, Attr, s)$. So, DEC(1) is satisfied because $LNCC_S(1)$ contains these tuples (cf. Sect. 4.2). Since *c1* does not have any context, no additional elements need to be translated and rule (1) is marked *appropriate*. Moreover, its core match can be completed to a full match. Therefore, $r^{(1)}_{ST}$ is an applicable rule. As it is the only applicable

```
1  procedure GraphTriple evolve(inputGraphTriple: GraphTriple) {  // GT_in
2    global inputGraph: Graph = Translator.getInputGraph(inputGraphTriple);  // G_input
3    global translatedElements: ElementSet = inputGraph.getTranslatedElements();  // TX
4    global justRegardedElements: ElementSet = ∅;
5    inputValid: boolean = inputGraph.verifyConstraints(); // Def.8(1)/(3) satisfied?
6    outputGraphTriple: GraphTriple = translate(inputGraphTriple);  // produce GT_out
7    outputValid:boolean= Translator.getOutputGraph(outputGraphTriple).verifyConstraints();
8    translated: boolean = inputGraph.isCompletelyTranslated();
9    if (inputValid && outputValid && translated)
10     return outputGraphTriple; // successfully produced GT_out
11   else if (inputValid && !outputValid) // Def.10(3) violated!
12     throw TGGContainsIntegrityDestroyingProductionsError(outputGraphTriple, translated);
13   else throw InputGraphNotPartOfDerivableGraphTripleException( // user-error: ...
14     outputGraphTriple, inputValid, outputValid, translated);   // ...G_input ∉ L(GG_I)
15 }
16 procedure GraphTriple translate(graphTriple: GraphTriple) {
17   forall (node ∈ inputGraph) { translate(node); }
18   return graphTriple;
19 }
20 prodecure translate(n: Node) {
21   if (n ∈ translatedElements) return;
22   else if (n ∈ justRegardedElements)
23     throw CycleInRecursiveContextTranslationError(n, justRegardedElements);
24   else { justRegardedElements.add(n);
25     nodeLocallyTranslatable: boolean = false;
26     appropriateRules, applicableRules: PairSet<Rule, Match> = ∅;
27     candidateRules: RuleSet = select rules where r.primaryInputNode.type equals n.type;
28     forall (rule ∈ candidateRules) { // collect appropriate rules and core matches
29       compute core matches of rule in inputGraph with n as primary node;
30       forall (cm ∈ core matches) { // Def.15(2): g'_I(R_I\L_I) ∩ TX = ∅ satisfied!
31         if (not isDECSatisfied(n, join(cm.toBe, cm.context))) // g'_I(R_I\L_I) ∪ g'_I(L_I)
32           { continue; } // do not translate context if Def.15(DEC(1)) would be violated
33         forall (contextNode ∈ context elements of core match)
34           { translate(contextNode); } // recursively translate required context
35         if (all context elements of core match are translated)
36           { appropriateRules.add(rule, cm); } // Def.15(1): g'_I(L_I) ⊆ TX satisfied!
37     } } // end of appropriate rule at core match with n as primary node calculation
38     forall ((rule, cm) ∈ appropriateRules) { // collect rules applicable at full match
39       // reassure Def.15(DEC(1)): may be violated due to competing context translation
40       if (not isDECSatisfied(n, cm.toBe)) { continue; }
41       nodeLocallyTranslatable = true;  // now n must be translatable due to Def.13
42       if (cm can be completed in other domains and NACs in output domain don't block)
43         { applicableRules.add(rule, full match); }
44     } // end of applicable rule at full match with n as primary node calculation
45     if (applicableRules.isEmpty()) { // node is not translatable
46       if (nodeLocallyTranslatable) // match could not be completed in other domain(s)
47         throw LocalCompletenessCriterionError(n, appropriateRules); // Def.13 violated!
48       else
49         throw InputGraphNotPartOfDerivableGraphTripleException(n); // G_input ∉ L(GG_I)
50     }
51     if (not applicableRules.matches->forAll(m1, m2 | m1 <> m2 implies
52       m1.cm.toBe = m2.cm.toBe)) throw CompetingCoreMatchesError(n, applicableRules);
53     select one rule/match pair from applicableRules;
54     apply rule at match;  // evolve GT ⇝ GT' with r_IO@g_IO
55     translatedElements.add(elements of inputGraph translated by rule); // g'_I(R_I\L_I)
56     justRegardedElements.remove(n);
57 } }
58 procedure boolean isDECSatisfied(node: Node, toBeTranslated: ElementSet) {
59   translatedElements' = translatedElements ∪ toBeTranslated; // TX' (cf. Def.15(3))
60   select all incident edges e of node where (e ∉ translatedElements')
61                                        and (s(e) or t(e) ∈ translatedElements')
62   forall (e ∈ selected incident edges)
63     { if (not (DEC(1) satisfied for e)) { return false; } } // ensure Def.15(DEC(1))
64   return true; // all edges translatable
65 }
```

Algorithm 1. Algorithm that handles NACs and checks for dangling edges.

rule it is applied at the complete match which translates node *c1* and creates a corresponding table in the target domain. The algorithm proceeds translating by selecting node *a3* from the set of remaining nodes $\{c2, a1, a2, a3, a4\}$. FGT rules (3), (4), and (5) are candidate rules for translating *a3*. But a core match, which requires *c2* as context, exists only for rules (3) and (4). First, the core match of rule $r_{ST}^{(3)}$ is examined. DEC(1) is satisfied for $inc(a3) = \{e7\}$ as $(Attr, Precedes, Attr, s) \in LNCC_S(1)$. Now, the context *c2* is translated by a recursive call to *translate(Node)*. The candidate rules for translating *c2* are $r_{ST}^{(1)}$ and $r_{ST}^{(2)}$. The algorithm randomly selects $r_{ST}^{(2)}$ to be checked first. DEC(1) is not violated (cf. Sect. 4.2) and the context of *c2* (i.e., *c1*) is already translated. The algorithm does not translate *c1* again because it notices that *c1* is already translated. After marking rule (2) as appropriate, candidate rule $r_{ST}^{(1)}$ is checked. It would violate DEC(1) and therefore it is not added to the set of applicable rules. As the match of $r_{ST}^{(2)}$ can be completed it is the only applicable rule and used to translate *c2*. The algorithm returns from the recursion and resumes in the context of *a3*. It marks $r_{ST}^{(3)}$ as appropriate and proceeds with $r_{ST}^{(4)}$ which is also appropriate. Though, the only applicable rule is $r_{ST}^{(3)}$ because, contrary to $r_{ST}^{(4)}$, its match can be completed. Consequently, both locally-applicable rules $r_{ST}^{(3)}$ and $r_{ST}^{(4)}$ are *disjoint applicable* in this case (cf. discussion in Sect. 4.1). Therefore, $r_{ST}^{(3)}$ is used to translate *a3*. Right now, the remaining nodes are $\{a1, a2, a4\}$ and the current graph triple looks like depicted in Fig. 10 (b). The following translation steps are rather similar to the preceding steps so we will abbreviate the explanation. Next, attribute *a1* is translated. Its context $\{c1\}$ is already translated. Rule candidates with a core match are $r_{ST}^{(3)}$ and $r_{ST}^{(4)}$, which both satisfy DEC(1), but $r_{ST}^{(3)}$ is blocked due to the NAC in the output domain. FGT rule (4) is applicable as a match of the LHS is found and the NAC does not block. Finally, attributes *a2* and *a4* are translated in this order. Their context is already translated and the rule candidates are $r_{ST}^{(3)}$, $r_{ST}^{(4)}$, and $r_{ST}^{(5)}$. FGT rules (3) and (4) are neglected as their application both would violate DEC(1) for $inc(a2) = \{e4\}$ and $inc(a4) = \{e7\}$ because $(Attr, Precedes, Attr, st) \notin LNCC_S(1)$. Hence, application of $r_{ST}^{(5)}$ would not violate DEC(1). Therefore, both attributes *a2* and *a4* are translated by $r_{ST}^{(5)}$.

So, the algorithm has successfully translated all nodes and edges of the input graph to a corresponding output graph. The sequence of applied FGT rules $SEQ(r_{ST}) = (r_{ST}^{(1)}@\emptyset, r_{ST}^{(2)}@c1, r_{ST}^{(3)}@c2, r_{ST}^{(4)}@c1, r_{ST}^{(5)}@c1, r_{ST}^{(5)}@c2)$ constructed in this example translates the primary nodes in this order: $(c1, c2, a3, a1, a2, a4)$. In conjunction with the also constructed correspondence graph a graph triple GT_{out} was produced which is equivalent to graph triple GT_6 (cf. Fig. 3 (a)) that was derived from TGG production sequence SEQ_6 (cf. Sect. 2.3) starting with the empty graph triple. Therefore, the FGT sequence exactly mimics SEQ_6.

The next theorems state that translators based on Algorithm 1 are efficient as well as consistent and complete with respect to their TGG if the algorithm never aborts for any given valid input graph. If the algorithm aborts then either

$G_{input} \notin \mathcal{L}(GG_I)$ or the TGG specification is erroneous, i.e., does not satisfy the conditions stated throughout this contribution.

Theorem 2. *Efficiency of Graph Translation.*
Algorithm 1 has worst case runtime complexity of $O(n^k)$ with n being the number of nodes of G_{input} and k being a constant that depends on the regarded TGG.

Proof.
Sketch:

(1) The algorithm just loops through the set of all n nodes of the input graph; the implicit reordering of the translation of input graph elements in the loop for not yet translated context elements of a just regarded graph element does not affect its runtime complexity.
(2) The book keeping overhead of the algorithm is neglectible and the execution time for basic graph operations like traversing an edge or creating a new graph element is bounded by a constant (otherwise we should add a logarithmic or linear term depending on the implementation of the underlying graph data structure).
(3) The worst case execution time of all needed rules applied to a given (primary) input graph node is $(n+n')^{k-1}$, where n' is the number of nodes of the output graph, and k is the maximum number of elements of any applicable rule. In the worst case the match of the primary node is extended by testing all possible $(n+n')^{k-1}$ permutations of source/target graph elements.
(4) Furthermore, $n' \leq c * n$ for a given constant c that is the maximum number of new nodes of the output component of any TGG production. □

In the case of TGG_{CDDS}, the complexity of a forward translation is $O(n^2)$ for the following reasons: The worst case execution time of its rules (cf. Fig. 4) is $O(n') \leq O(n)$ due to the fact that rules (1), (2), and (3) have a constant execution time, whereas rules (4) and (5) have to determine the last column node of a table node. Assuming that all nodes of the output graph are columns of the just regarded table $n' \leq n$ nodes have to be inspected in the worst case.

Theorem 3. *Consistency of Graph Translation.*
Let $G_I \in \mathcal{L}(TG_I, \mathcal{C}_I)$ be an input graph (either G_S or G_T) and G_O be an output graph (either G_T or G_S). If
$FGT(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset) = (G_S \leftarrow G_C \rightarrow G_T)$ and
$BGT(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T) = (G_S \leftarrow G_C \rightarrow G_T)$, respectively,
is a not aborting complete translation of G_I with Algorithm 1 then:
(1) $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ and (2) $G_O \in \mathcal{L}(TG_O, \mathcal{C}_O)$.

Proof.
Sketch:

(1) $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ is a direct consequence of Theorem 1 and the fact that $G_I \in \mathcal{L}(TG_I, \mathcal{C}_I)$. As a consequence the simulated application of TGG productions without NACs in the input domain does not have any effect concerning the applicability of translation rules.

(2) The behavior of translation rules on the output side is identical with the behavior of the related TGG production: i.e., a rule finds a match on the output side iff the related TGG production has the same match.

$G_O \in \mathcal{L}(TG_O, \mathcal{C}_O)$ is then a direct consequence of (1). □

Theorem 4. *Completeness of Graph Translation.*

Let $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ *and let us assume that the execution of Algorithm 1 does not abort with any error. Then, we can guarantee that graphs* G^*_C, G^*_T *and* G^*_S, G^*_C, *respectively, exist such that:*

$FGT(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset) = (G_S \leftarrow G^*_C \rightarrow G^*_T) \in \mathcal{L}(TGG)$ *and*

$BGT(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T) = (G^*_S \leftarrow G^*_C \rightarrow G_T) \in \mathcal{L}(TGG)$, *respectively;*

i.e., the algorithm terminates without throwing any exception.

Proof.

(by induction) Sketch:

Let $GT_{out} \in \mathcal{L}(TGG)$ be a graph triple that has been derived using a sequence of derivation steps $SEQ^n_{i=1}(p_i) = ((p@g)_1, \ldots, (p@g)_n)$ of length n and let $SEQ^n_{i=1}(r_{I,i}) = ((r_I@g_I)_1, \ldots, (r_I@g_I)_n)$ be the projection of the regarded sequence of graph triple derivation steps on its input graph. Furthermore, let $SEQ^m_{i=0}(r_{IO,i})$ with $0 \leq m \leq n$ be the sequence of the first m translation rule applications $((r_{IO}@g_{IO})_1, \ldots, (r_{IO}@g_{IO})_m)$ generated by the algorithm that exactly mimics the derivation of GT_{out}.

Case 1, $m = 0$: A translation rule sequence of length 0 trivially mimics the derivation of the empty graph triple $GT_\emptyset$.

Case 2, $0 < m < n$: We have to show that the algorithm extends the given sequence of rule applications of length m to a sequence of length $m+1$ such that it simulates either the original sequence of TGG productions $SEQ(p)$ or a slightly modified sequence $SEQ(p^*)$ that still generates the same input graph. Let $(v_I)_i$ be the primary node of the input graph of each rule application $(r_{IO}@g_{IO})_i$ and TGG production application $(p@g)_i$ with $1 \leq i \leq m$. Let v be the next to-be-translated primary node which is selected by the algorithm. Furthermore, we assume that the algorithm has already translated successfully the context nodes of all rules that might be able to translate node v.

Case 2.1, $v = (v_I)_{m+1}$: Due to the fact that the algorithm does not throw a CompetingCoreMatchesError we can safely assume that there exists at most one set of translation rules with the same to-be-translated elements in their core match including node v. Furthermore, we know that there exists at least one rule with $p^-_{I,id}$ (that is the input component of the translation rule derived from p_{m+1}) that matches node $v = (v_I)_{m+1}$. The local completeness criterion (cf. Def. 13) guarantees that the algorithm finds a TGG production application $p^*@g^*$ that corresponds to one of the translation rules r^*_{IO} that is able to handle the translation of the selected node v. Applying Def. 13 multiple times we can generate a new sequence of TGG production applications $SEQ^n_{i=1}(p^*_i)$ such that:

$1 \leq i \leq m$: $(p^*@g^*)_i = (p@g)_i$

$i = m + 1$: $(p^*@g^*)_i = p^*@g^*$

$m + 1 < i \leq n$: $(p^*@g^*)_i$ is a new production application that mimics $(p@g)_i$

As a consequence the algorithm is able to create a sequence of translation steps $SEQ(r^*_{IO})$ of length $m+1$ that has the same properties as the given sequence of translation steps $SEQ(r_{IO})$ of length m w.r.t. the new sequence $SEQ^n_{i=1}(p^*_i)$ that replaces $SEQ^n_{i=1}(p_i)$.

Case 2.2, $v \neq (v_I)_{m+1}$: Due to the fact that the selected node v is not yet translated and that $(v_I)_1, ..., (v_I)_n$ is the complete set of all primary nodes of the given input graph (generated by the given sequence of TGG production applications) there exists an index $k > m+1$ such that $v = (v_I)_k$. Let $(p^-_{I,id})_k$ be the input component of the translation rule derived from $(p@g)_k$. We know that all context nodes potentially required by $(p^-_{I,id})_k$ are already translated. Again relying on the fact that the algorithm does not throw any error and on Def. 13 we know that a rule r^*_{IO} exists, derived from a production p^*, which is able to translate the given primary node v. Using the same line of arguments as in case 2.1 we can construct a new sequence of TGG productions p^* of length n with the same properties as listed above. As a consequence the algorithm is again able to create a sequence of translation steps $SEQ(r^*_{IO})$ of length $m+1$ that has the same properties as the given sequence of translation steps of length m.

Case 3, $m = n$: The translation rule sequence mimics the complete derivation of the input graph, i.e., generates a valid translation into a graph triple GT^*_{out} that has the same input graph as GT_{out} but may have different correspondence and output graphs then GT_{out}. □

The consequence of the proof sketches is as follows. If we are able to show for a given TGG that derived translators never abort with an error then:

(1) The presented algorithm can be executed efficiently (polynomial complexity) as long as the matches of all translation rules can be computed efficiently.
(2) Forward and backward translation results are consistent, i.e., do only produce graph triples that belong to the language of the regarded TGG.
(3) Forward and backward translations will always produce a result for a given input graph if the language of the regarded TGG contains a graph triple that has this input graph as a component.

Finally, our running example shows that in the general case the result of a graph translation is not uniquely determined up to isomorphism, i.e., sets of TGG productions needed in practice often do not satisfy any (local) confluence criteria. Therefore, it was of importance to develop an efficiently working graph translation algorithm that does not rely on (local) confluence criteria of TGG productions or translation rules, but nevertheless fulfills the initially presented expressiveness, consistency, and completeness properties, too!

6 Related Work

Based on the characterization of "useful" TGGs in Sect. 2 we proposed extensions to triple graph grammars in Sects. 3, 4, and 5. Now, we are prepared to evaluate and assess various forms of declarative bidirectional model or graph transformations that have been published in the past.

The first TGG publication [1] introduced a rather straight-forward construction of translators. It relied on the existence of graph grammar parsing algorithms with exponential worst-case space and time complexity. As a consequence a first generation of follow-up publications [13,14] all made the assumption that the regarded graphs have a dominant tree structure and that the components of a TGG production possess one and only one primary node. Based on these assumptions an algorithm is used that simply traverses the tree skeleton of an input graph node by node and selects an arbitrary matching FGT/BGT rule for a regarded node that has a node of this type as its primary node. This algorithm defines translation functions that are neither consistent nor complete in the general case. Both properties are endangered by the fact that the selected tree traversal order does not guarantee that rules are applied in the appropriate order. It may happen that the application of a rule fails because one of its context nodes has not been processed yet or that a rule is applied despite of the fact that one of its context nodes has not been matched by another rule beforehand.

As a consequence, [8] introduces an algorithm that still relies on a tree traversal, but keeps track of the set of already processed nodes and uses a waiting queue to delay the application of rules if needed. This algorithm defines consistent translators, but has an exponential worst-case behavior concerning the number of re-applications of delayed rule instances. Another class of TGG approaches (cf. [15]) attacked the rule ordering problem in a rather different way. These approaches introduce a kind of controlled TGGs, where each rule explicitly creates a number of child rule instances that must be processed afterwards. Thus, one of the main advantages of a rule-based approach is in danger that basic rules can be added and removed independently of each other and that it is not necessary to encode a proper graph traversal algorithm explicitly.

All publications mentioned so far refrained from the usage of NACs that where introduced in [16] in the context of model transformation approaches based on graph transformation. Some of them even argued that NACs cannot be added to TGGs without destroying their fundamental properties! But, rather recently some application-oriented TGG publications simply introduced NACs without explaining how derived translation rules and their rule application strategies have to be adapted precisely. The publications even give the reader the impression that NACs can be evaluated faithfully on a given input graph without regarding the derivation history of this graph with respect to its related TGG. [17], e.g., explicitly makes the proposal to handle complex graph constraints in this way, whereas [18] and [4] ignore the problems associated with the usage of NACs completely. Despite these TGG approaches that already introduced NACs to TGGs without a guarantee for consistency and completeness, we proposed translators in [5] that guarantee consistency. Nevertheless, we could not guarantee completeness of these translators.

We have to reference [19] as the first publication that studied useful properties of translators including "invertibility" from a formal point of view. The authors of this paper are interested in pairs of translation relations that are inverse to each other. As a consequence they have to impose hard restrictions on TGGs in order

to be able to construct their proofs. Furthermore, [19] has a main focus on consistency, whereas efficiency, expressiveness, and completeness are out-of-scope. In addition, [19] extends the concept of triple graphs based on simple graphs to triple graphs based on typed, attributed graphs. Follow-up publications (e.g., [20] and [10]) then introduced NACs in an appropriate way and proved that translators may be derived from a TGG with NACs that are compatible with their TGG. Unfortunately, both [20] and [10] trade efficiency for completeness. That is, neither [20] nor [10] present an algorithm that is able to find an appropriate sequence of translation rules in polynomial time which is necessary to create efficiently working translators. Compared to both approaches, we showed here that we are able to derive compatible translators—from a precisely defined subset of TGGs with NACs—which are still efficient.

Other bidirectional model/graph translation approaches either suffer from similar deficiencies or circumvent the efficiency versus completeness tradeoff problem as follows: QVT Relational [3] as a representative of this sort of model transformation approaches simply applies all matches of all translation rules to a given input model in parallel and merges afterwards elements of the generated output model based on key attributes. This approach is rather error-prone and requires a deep insight of the QVT tool developer as well as its users how rules match and interact which each other. As a consequence, [21] shows that today existing QVT Relational tools may produce rather different results when processing the same input. For a more comprehensive survey of bidirectional transformation approaches the reader is referred to [22].

7 Conclusion and Future Work

In this contribution we presented a "useful" class of triple graph grammars together with translators that comply to the four design principles stated in the *"Grand Research Challenge of the Triple Graph Grammar Community"* introduced in [5]: the development of a consistent, complete, and efficient graph translation algorithm for a hopefully still sufficiently expressive class of triple graph grammars (TGGs). For this purpose we combined (a) restrictions for negative application conditions of TGG productions with (b) a dangling edge condition for graph translation rules that was inspired by "look ahead" concepts of parsing algorithms. As a consequence, graph translators derived from the thus restricted class of TGGs no longer have to take care of rule application conflicts by either using a depth-first backtracking parsing algorithm or a breadth-first computation of all possible derivations of a given input graph. Therefore, the presented new graph translation algorithm has a polynomial runtime complexity of $O(n^k)$ for a rather small k in practice that is determined by the worst-case complexity of computing matches for all needed graph translation rules.

Promising directions for future work are, e.g., the adaptation of the confluence checking algorithms of AGG [23], which are based on critical pair analysis, as well as the model checking approach for graph grammars of GROOVE [24] to the world of TGGs. Confluence checking techniques should offer the right

means for the detection and classification of potential rule application conflicts at compile time. In this way we would be able to guarantee already at compile time that a graph translator derived from a specific class of TGGs will not stop its execution with an error instead of generating an existing output graph for a given input graph. Furthermore, constraint verification techniques of GROOVE should allow to check the here introduced requirements already at compile time: (a) TGG productions never create graph triples that violate graph constraints of the related schema, (b) NACs are only used to block graph modifications that would violate a graph constraint, (c) TGG productions never repair constraint violations by rewriting an invalid graph into a valid graph, and (d) TGGs fulfill the local completeness criterion. Until then, TGG developers have to design and test their TGGs carefully such that TGG productions do not violate the presented conditions of integrity-preserving productions. Moreover, the presented algorithm has to be extended to cope with secondary elements and to perform incremental updates in order to also synchronize changes in source and target domain models. In addition, the limited class of TGGs with NACs presented in this contribution has to be enlarged but compatibility and efficiency properties of derived translators have to be ensured. We are currently evaluating the here presented class of TGGs with NACs in research cooperations with industrial partners, where TGGs are used to ensure consistency of design artifacts. Time will show whether our claim is true that the here introduced new class of TGGs is still expressive enough for the specification of a sufficiently large class of bidirectional model/graph translations that are needed in practice.

References

1. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. Addison-Wesley, Reading (2003)
3. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.0. (April 2008), http://www.omg.org/spec/QVT/1.0/
4. Taentzer, G., et al.: Model Transformation by Graph Transformation. In: Model Transformation in Practice (MTiP 2005), Workshop at MODELS 2005 (2005)
5. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
6. Bruel, J.-M. (ed.): Satellite Events at the MoDELS 2005 Conference. LNCS, vol. 3844. Springer, Heidelberg (2006)
7. Zündorf, A.: Rigorous Object Oriented Software Development. University of Paderborn, Habilitation Thesis (2001)
8. Königs, A.: Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation. PhD thesis, TU Darmstadt (2009)
9. Rekers, J., Schürr, A.: Defining and Parsing Visual Languages with Layered Graph Grammars. Journal of Visual Languages and Computing 8(1), 27–55 (1997)

10. Ehrig, H., Ermel, C., Hermann, F., Prange, U.: On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 241–255. Springer, Heidelberg (2009)
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Series. Springer, Heidelberg (2006)
12. Heckel, R., Cherchago, A.: Structural and behavioural compatibility of graphical service specifications. J. Log. Algebr. Program. 70(1), 15–33 (2007)
13. Lefering, M.: Software document integration using graph grammar specifications. In: 6th International Conference on Computing and Information. Journal of Computing and Information, vol. 1, pp. 1222–1243 (1994)
14. Jahnke, J., Schäfer, W., Zündorf, A.: A design environment for migrating relational to object oriented database systems. In: 12th International Conference on Software Maintenance (ICSM 1996), pp. 163–170 (1996)
15. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
16. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Fundamenta Informaticae 26(3-4), 287–313 (1996)
17. Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn, Germany (2007)
18. Grunske, L., Geiger, L., Lawley, M.: Graphical Specification of Model Transformations with Triple Graph Grammars. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 284–298. Springer, Heidelberg (2005)
19. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
20. Ehrig, H., Hermann, F., Sartorius, C.: Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. ECEASST 18 (2009)
21. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. Software and Systems Modeling (2009)
22. Czarnecki, K., Foster, J., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional Transformations: A Cross-Discipline Perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)
23. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
24. Rensink, A.: Explicit State Model Checking for Graph Grammars. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 114–132. Springer, Heidelberg (2008)

Controlling Reuse in Pattern-Based Model-to-Model Transformations

Esther Guerra[1], Juan de Lara[2], and Fernando Orejas[3]

[1] Universidad Carlos III de Madrid, Spain
eguerra@inf.uc3m.es
[2] Universidad Autónoma de Madrid, Spain
Juan.deLara@uam.es
[3] Universitat Politècnica de Catalunya, Spain
orejas@lsi.upc.edu

Abstract. Model-to-model transformation is a central activity in Model-Driven Engineering that consists of transforming models from a source to a target language. Pattern-based model-to-model transformation is our approach for specifying transformations in a declarative, relational and formal style. The approach relies on patterns describing allowed or forbidden relations between two models. These patterns are compiled into operational mechanisms to perform forward and backward transformations.

Inspired by QVT-Relations, in this paper we incorporate into our framework the so-called check-before-enforce semantics, which checks the existence of suitable elements before creating them (i.e. it promotes reuse). Moreover, we enable the use of *keys* in order to describe when two elements are considered equal. The presented techniques are illustrated with a bidirectional transformation between Web Services Description Language and Enterprise Java Beans models.

1 Introduction

Model-Driven Engineering (MDE) [28] proposes the construction of software systems using models as primary artefacts. In this paradigm, models are used to specify, reason, generate code, document, test, analyse and maintain the final application. Hence, model transformation becomes a key enabling technology for MDE, and is being subject of intensive research nowadays. The Model-Driven Architecture [17] (MDA) is a particular incarnation of MDE promoted by the OMG, which proposes the use of its standard languages, like MOF for metamodelling and QVT [23] (Query/View/Transformation) for transformations.

Model-to-Model (M2M) transformation involves transforming models from a source to a target language. In the context of MDE, M2M transformations are used e.g. to migrate between language versions, to refine a model, or to transform a model into a semantic domain for analysis. Several usage scenarios can be identified. Source-to-target (resp. target-to-source) transformations assume the existence of a source (resp. target) model and create a target (resp. source) model from scratch. *Incremental* transformations optimize the former, so that

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 175–201, 2010.

if the source (resp. target) model is changed after being transformed, the target (resp. source) is updated but not regenerated. A further step is *model synchronization*, where both models can be modified at any time, and the changes are propagated to the other model to recover consistency. Hence, a sensible approach is to define a unique specification establishing when two models are consistent, and then generate specific lower-level *operational* mechanisms to solve the scenario of interest. This has the advantage that the transformation is specified only once, but it requires using a bidirectional, declarative style of specification. Moreover, the synthesis of operational mechanisms may imply complex algebraic manipulations of the declarative attribute conditions appearing in the transformation specification.

Even though many transformation languages have been proposed in the literature [23,25], there is a need for expressive, high-level, and formal languages able to precisely express the M2M consistency problem and enabling the analysis of the transformation specifications. Following the ideas of [25], but aimed at a relational style of specifications in the lines of [23,27], in [4] we developed a new approach for specifying M2M transformations. The approach is based on *patterns* describing positive or negative conditions that are to be satisfied by two models in order to be considered consistent. Patterns have a high-level (i.e. independent of the operational mechanism), algebraic semantics enabling the decision of whether two models are consistent, or to find the discrepancies with respect to the specification. These patterns are then compiled into operational mechanisms, based on triple graph grammar rules [4,12], but in which no algebraic manipulation of attribute formulae is necessary.

We believe our framework can be used to formalise other transformation languages, especially QVT-Relations (QVT-R). The purpose of this paper is to advance in this direction. With this aim, we extend our previous works [4,12,21] by bringing into our framework two concepts of QVT-R: the Check-Before-Enforce (CBE) semantics and the *keys*. CBE semantics is a way to promote element reuse in transformations, and to enable many-to-one relations between elements across models. In particular, before creating an element, it is checked whether an existing one can be reused. Keys allow specifying when two elements are considered equal. Moreover, in order to promote the use of our techniques in MDE, we propose a way to enrich the transformation specification with integrity constraints of the source and target meta-models, in particular the association cardinality constraints. Finally, we extend our patterns by allowing *abstract objects*. These features are illustrated through a transformation between Web Service Description Language [29] (WSDL) models and Enterprise Java Beans [19] (EJBs).

Paper organization. Section 2 introduces related work. Section 3 presents the case study we will use throughout the paper. Section 4 recalls the necessary background for subsequent sections. Section 5 reviews our notion of patterns, and Section 6 the generation of source-to-target and target-to-source operational mechanisms. Then, Section 7 incorporates the CBE semantics and keys into our framework. Section 8 presents further details of the case study and, finally,

Section 9 ends with the conclusions. An appendix presents some of the proofs of the main claims and propositions.

2 Related Work

Bidirectional transformation languages are receiving increasing attention in MDE, as they are able to capture consistency relations between two models in a direction-independent way. In this approach, a unique specification is used to derive operational mechanisms solving the different synchronization scenarios mentioned in the introduction (see also [16]).

A prominent example of this kind of languages is QVT-R [23], a part of the QVT family of transformation languages sponsored by the OMG. A QVT-R specification is made of relations, each consisting of two or more domains (i.e. models). Relations can be *top* or *non-top* level, and include *when* and *where* clauses that may be used to express dependencies between relations. The execution of a transformation requires that all its top-level relations hold, whereas the non-top level ones only need to hold when invoked from the *where* section of other relations. The standard specifies that QVT-R models are enforced by its compilation into QVT-core, a lower-level language. While QVT-R has no explicit notion of traces (i.e. relations between the model elements involved in the transformation), the compilation to QVT-core creates them automatically.

In [1], transformations are expressed through declarative relations made of positive patterns, heavily relying on OCL constraints, but no operational mechanism is given to enforce such relations. In BOTL [2], the mapping rules use a UML-based notation that allows reasoning about applicability or meta-model conformance. In our approach we can reason both at the specification and operational levels. In [6], the authors rely on completely relational transformation units and infer the order of execution by studying their dependencies. They use attribute grammars as (uni-directional) transformation language. This kind of grammars is made of textual relations where the order of execution of rules is not given, but it is automatically calculated in accordance with the dependency relations that arise between attributes. In the MTF language from IBM [18], transformations are made of textual relations expressed in RDL (the *Relations Definition Language*) that do not impose a direction of the transformation, but this is selected when invoking the transformation engine. Similar to QVT-R, MTF relations must be invoked from other relations in order to be executed, whereas in our approach we *query* the trace model.

TGGs [25] formalize the synchronized evolution of two graphs through declarative rules. The language spawned by these rules contains the pairs of models considered consistent. From this specification, low-level operational rules are derived to solve different synchronization scenarios. Interestingly, our patterns define a language of valid consistent models by means of constraints instead of rules (even though the operational mechanisms are implemented through operational rules) and hence we admit negative constraints too. The work in [15] included in this volume improves previous works on TGGs by considering TGG

schemas (meta-model triples in our jargon) with so called monotonic constraints (which if satisfied by a model, are satisfied by any submodel). These constraints are similar to our notion of N-patterns, but our N-patterns are not necessarily monotonic. Also, they propose a guiding mechanism for applying the operational rules by a so called *dangling edge condition*, which before applying one transformation rule checks if some edge in the source will not get translated. In our case, we assume that not every element in the source needs to be translated, but the fact that we generate several rules for each pattern permits obtaining all valid target models, if more than one exists [21]. An attempt to bridge TGGs and QVT-R is [10], where QVT-R is both compiled into operational TGGs (instead of using QVT-core) and translated into declarative TGGs.

An interesting issue in these languages is how to handle and express object reuse. This aspect has been tackled in QVT-R by the CBE semantics, where the operational mechanism checks which objects exist and can be reused before creating them. In order to specify when two elements are considered equal, one can set *keys* (similar to keys in databases). Reuse has to be handled explicitly in TGGs by including the objects to be reused in the left-hand side (LHS) of the declarative rules. Up to now, our patterns followed a similar approach by defining the objects to reuse as positive pre-conditions. It is interesting however to decouple the specification of the reusing policy (keys in QVT) from the specification of the transformation itself, which potentially leads to more flexible and reusable transformations. In this paper we incorporate these ideas into our framework.

3 The Example Case Study

In this section we introduce the case study that we will use throughout the paper, namely the transformation between WSDL documents and EJBs, both represented as models. WSDL [29] is an XML-based language for describing web services, endorsed by the W3C. Here we use the last version 1.1, which is the most widely used by tools. Fig. 1 shows a simplified meta-model for WSDL we have developed taking as a basis the XML syntax described in [29].

A WSDL model includes the definition of services as collections of network endpoints, or ports (class `Port` in the meta-model). Ports and messages are described in an abstract way through classes `PortType` and `Message`, independently from their concrete usage. Then, a *binding* provides the concrete information (addresses, protocols – normally HTTP – and so on) to use the services through their ports. The binding is usually done through SOAP [30], although for simplicity we have omitted the classes for the binding from the meta-model. A `PortType` defines a number of operations (similar to functions in programming languages) that the service exposes, modelled by class `Operation`. There are four types of operation, defining a protocol for exchanging messages. For example, while operations of type `OneWay` just receive one message, `RequestResponse` operations in addition send a response back. The operations refer to the messages involved,

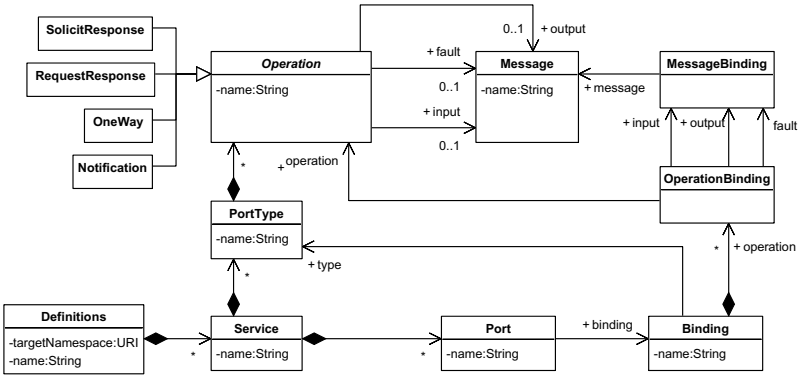


Fig. 1. WSDL meta-model (simplified)

either input, output or fault. A `Message` has a structure made of several logical parts, omitted here for simplification.

Enterprise JavaBeans (EJBs) [24] is a Java API that defines a component architecture to build server-side enterprise applications. Its specification provides a number of services commonly found in these applications, like persistence, transaction processing, concurrency control, security and exposing business methods as web services, among others. Fig. 2 shows a simplification of its meta-model we have developed taking [19] as a basis. An EJB container (class `EJBJar`) can hold a number of beans (i.e. Java components), the most important types of which are `Session` and `Entity`. The former are distributed objects that can have a state or not, depending on whether their attribute `sessionType` takes the value `stateful` or `stateless`, respectively. Stateful beans keep track of the calling process through a session, and hence a different bean instance is created for each customer. On the contrary, stateless beans enable concurrent access. Entity beans (class `Entity`) represent persistent data maintained in a database[1]. For simplicity, we have omitted the details of this kind of beans.

EJBs are deployed in an application server. Each EJB has to provide a Java implementation class, and two interfaces called *Home* and *Remote*. The meta-model in Fig. 2 contains a high-level Java meta-model that reflects the dependency of EJBs to Java.

In our case study, we are interested in specifying a bidirectional transformation between WSDL and EJB models. This is useful as, when building EJB applications, it is sometimes needed to expose them as web services, and hence to generate a WSDL file with the service description. The generated operational (backward) transformation would do this automatically. The opposite is also common, sometimes a WSDL file with the description of a service needs to be implemented. The generated operational (forward) transformation would

[1] We use the EJB1.1 specification; in the EJB3.0 specification Entity Beans were superseded by the Java Persistence API.

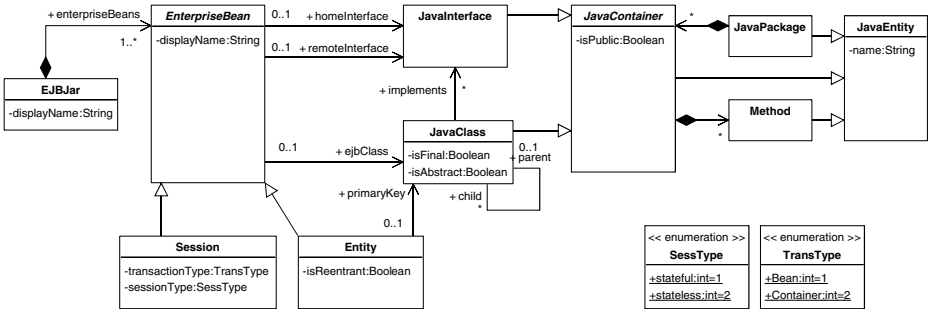


Fig. 2. EJB meta-model (simplified)

synthesize an EJB model containing skeletons of the necessary Java classes and interfaces. There are already available tools that perform these tasks. For example, the Oracle Containers for J2EE (OC4J) [20] has a tool called `wsdl2ejb` that generates an EJB from a WSDL file. Similarly, the IBM Websphere application server [13] provides the `EJB2WebService` tool to create a web service (including the WSDL file) from EJBs. Note however that both tools are not incremental and overwrite existing files. Our method has the potential to be incremental and moreover it would generate both tools starting from a single specification.

4 Preliminaries

In this section we introduce the basic theoretical concepts (triple graphs and constraint triple graphs) that we will use in our M2M specification language.

In order to perform M2M transformations, it is useful to consider structures made of a source and a target model, related through a trace model. This structure is called *triple graph* [25]. As we can provide nodes and edges in graphs with attributes and types (called E-graphs in [8]), models can be naturally encoded with graphs. An E-graph is a tuple $G = (V^G, D^G, E^G, E^G_{NA}, E^G_{EA}, (src^G_j, tar^G_j)_{j\in\{G,NA,EA\}})$, where V^G and D^G are sets of graph and data nodes, E^G is a set of graph edges, E^G_{NA} and E^G_{EA} are sets of edges modelling attributes for both nodes and edges, functions $src^G_G\colon E^G \to V^G$ and $tar^G_G\colon E^G \to V^G$ are the graph edge source and target functions, $src^G_{NA}\colon E^G_{NA} \to V^G$ and $tar^G_{NA}\colon E^G_{NA} \to D^G$ are the source and target functions for node attributes, and $src^G_{EA}\colon E^G_{EA} \to E^G$, $tar^G_{EA}\colon E^G_{EA} \to D^G$ are the functions for edge attributes. Even though we use E-graphs in our triple graphs, any other type of graph could also be used. Graphs can be typed by a type graph TG (similar to a meta-model) [8] becoming objects of the form $(G, type\colon G \to TG)$, where *type* is a typing function.

Hence, triple graphs are made of three graphs: source (S), target (T) and correspondence (C). Nodes in the correspondence graph relate nodes in the source and target graphs by means of two graph morphisms [7].

Definition 1 (Triple Graph and Morphism). *A triple graph* $TrG = (S \xleftarrow{c_S} C \xrightarrow{c_T} T)$ *is made of three E-graphs* S, C *and* T *s.t.* $D^C = \emptyset$, *and two graph morphisms* c_S *and* c_T *called the source and target correspondence functions.*

A triple morphism $m = (m_S, m_C, m_T)\colon TrG^1 \to TrG^2$ *is made of three E-morphisms* m_X *for* $X = \{S, C, T\}$, *s.t.* $m_S \circ c_S^1 = c_S^2 \circ m_C$ *and* $m_T \circ c_T^1 = c_T^2 \circ m_C$, *where* c_S^x *and* c_T^x *are the correspondence functions of* TrG^x *(for* x*={1, 2}).*

Remark. The correspondence graph is restricted to be unattributed (i.e. $D^C = \emptyset$), but not necessarily discrete. This is so because otherwise, in general, we could not take c_S and c_T to be graph morphisms, as the conditions for attributes fail.

We use the notation $\langle S, C, T\rangle$ for a triple graph made of graphs S, C and T. Given $TrG = \langle S, C, T\rangle$, we write $TrG|_X$ for $X \in \{S, C, T\}$ to refer to a triple graph where only the X graph is present, e.g. $TrG|_S = \langle S, \emptyset, \emptyset\rangle$. Triple graphs and morphisms form the category **TrG**.

Example. The left of Fig. 3 shows a triple graph relating a WSDL model and an EJB model. The graph nodes are depicted as rectangles, and the data nodes in D^S and D^T as rounded rectangles. We only draw the used data nodes, as they may be infinite. We have represented the types of nodes after a semicolon. In fact, a triple graph is typed by a type triple graph (or meta-model triple [11]), where the typing morphism is a triple graph morphism. The right of the same figure shows the triple graph in UML notation, which we will use throughout the paper.

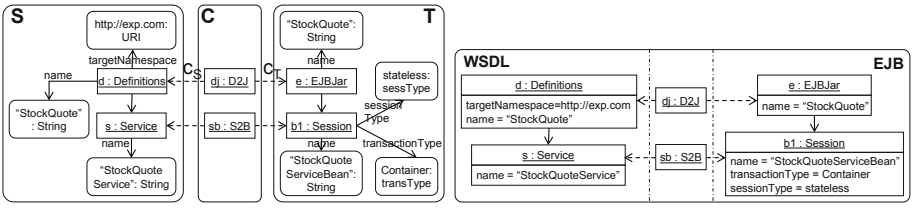


Fig. 3. Triple graph example in theoretical (left) and compact notations (right)

Next, we present the notion of *constraint triple graph* [12]. It will be used later as a building block of our patterns, as a way to express desired relations between the source and target models, and also in the left and right hand sides of the generated TGG operational rules. Constraint triple graphs are triple graphs attributed over a finite set ν of variables, and equipped with a formula on this set (i.e., a $\Sigma(\nu)-$formula, where Σ is a signature) to constrain the possible attribute values of source and target elements.

Definition 2 (Constraint Triple Graph). *Given an algebra* $\mathcal{A}$ *over signature* $\Sigma = (S, OP)$, *a* constraint triple graph $CTrG^{\mathcal{A}} = (TrG, \nu, \alpha)$ *consists of a triple graph* $TrG = \langle S, C, T\rangle$, *a finite set of S-sorted variables* $\nu = D^S \uplus D^T$ *(with* $\uplus$ *denoting disjoint union) and a* $\Sigma(\nu)-$*formula* α *in conjunctive or clausal form.*

Before defining morphisms between constraints, we need an auxiliary operation for restricting $\Sigma(\nu)-$formulae to a smaller set of variables $\nu' \subseteq \nu$. This will be useful for example when restricting a constraint triple graph to the source or target graph only. Thus, given a $\Sigma(\nu)$-formula α, its restriction to $\nu' \subseteq \nu$ is given by $\alpha|_{\nu'} = \alpha'$, where α' is like α, but with all clauses with variables in $\nu - \nu'$ replaced by **true**. Thus, for example $(x = 3) \wedge \neg(y = 7)|_{\{x\}} = (x = 3)$, as we substitute $\neg(y = 7)$ by **true**.

Given a constraint $CTrG^{\mathcal{A}} = (TrG, \nu, \alpha)$, we write α^S for the restriction to the source variables $\alpha|_{D^S}$, and α^T for the restriction to the target variables $\alpha|_{D^T}$. Given a variable assignment $\mathfrak{f}\colon \nu \to \mathcal{A}$, we write $\mathcal{A} \models_{\mathfrak{f}} \alpha$ to denote that the algebra $\mathcal{A}$ satisfies the formula α with the value assignment induced by $\mathfrak{f}$. Note that if $\mathcal{A} \models_{\mathfrak{f}} \alpha$, then $\mathcal{A} \models_{\mathfrak{f}} \alpha|_{\nu'}\ \forall \nu' \subseteq \nu$.

Morphisms between constraint triple graphs are made of a triple graph morphism and a mapping of variables (i.e. a set morphism). In addition we require an implication from the formula of the constraint in the codomain to the one in the domain, and also implications from the source and target restrictions of the formula in the codomain to the restrictions of the formula in the domain. This means that the formula in the domain constraint should be weaker or equivalent to the target (intuitively, the codomain may contain "more information").

Definition 3 (Constraint Triple Graph Morphism). *A constraint triple graph morphism $m = (m^{TrG}, m^{\nu})\colon CTrG_1^{\mathcal{A}} \to CTrG_2^{\mathcal{A}}$ is made of a triple morphism $m^{TrG}\colon TrG_1 \to TrG_2$ and a mapping $m^{\nu}\colon \nu_1 \to \nu_2$ s.t. the diagram to the left of Fig. 4 commutes, and $\forall \mathfrak{f}\colon \nu_2 \to \mathcal{A}$ s.t. $\mathcal{A} \models_{\mathfrak{f}} \alpha_2$, then $\mathcal{A} \models_{\mathfrak{f}} (\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)) \wedge (\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T)) \wedge (\alpha_2 \Rightarrow m^{\nu}(\alpha_1))$, where $m^{\nu}(\alpha)$ denotes the formula obtained by replacing every variable X in α by the variable $m^{\nu}(X)$.*

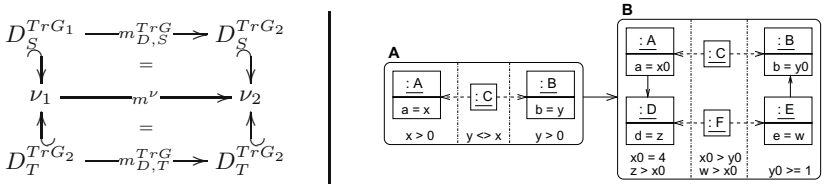


Fig. 4. Condition for **CTrG**-morphisms (left). Example (right).

Remark. Note that $\alpha_2 \Rightarrow m^{\nu}(\alpha_1)$ does not imply $\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)$ or $\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T)$. For technical reasons we require $(\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)) \wedge (\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T))$, as we need to build source and target constraint *restrictions* (see below) and obtain a morphism from the restricted constraint to the full constraint.

Example. The right of Fig. 4 shows a constraint triple graph morphism. Concerning the formula, if we assume some variable assignment $\mathfrak{f}\colon \nu_B \to \mathcal{A}$ satisfying α_B (i.e. s.t. $\mathcal{A} \models_{\mathfrak{f}} \alpha_B$), then such $\mathfrak{f}$ makes $\mathcal{A} \models_{\mathfrak{f}} [(x_0 = 4 \wedge z > x_0) \Rightarrow (x_0 > 0)] \wedge [(y_0 >= 1) \Rightarrow (y_0 > 0)] \wedge [(x_0 = 4 \wedge z > x_0 \wedge x_0 > y_0 \wedge w > x_0 \wedge y_0 >= 1) \Rightarrow (x_0 > 0 \wedge y_0 <> x_0 \wedge y_0 > 0)]$. Thus, the formula in A (the morphism domain) is weaker or equivalent to the formula in B (the morphism codomain).

From now on, we restrict to injective morphisms (for simplicity, and because our patterns are made of injective morphisms). Given Σ and $\mathcal{A}$, constraint triple graphs and morphisms form the category $\mathbf{CTrG}_{\mathcal{A}}$. As we will show later, we need to manipulate objects in this category through pushouts and restrictions. A pushout is the result of gluing two objects B and C along a common subobject A, written $B +_A C$. Pushouts in $\mathbf{CTrG}_{\mathcal{A}}$ are built by making the pushout of the triple graphs, and taking the conjunction of their formulae.

Proposition 1 (Pushout in $\mathbf{CTrG}_{\mathcal{A}}$). *Given the span of $\mathbf{CTrG}_{\mathcal{A}}$-morphisms $B^{\mathcal{A}} \xleftarrow{b} A^{\mathcal{A}} \xrightarrow{c} C^{\mathcal{A}}$, its pushout is given by $D^{\mathcal{A}} = (B +_A C, \nu_B +_{\nu_A} \nu_C, c'(\alpha_B) \wedge b'(\alpha_C))$, and morphisms $c' \colon B^{\mathcal{A}} \to D^{\mathcal{A}}$ and $b' \colon C^{\mathcal{A}} \to D^{\mathcal{A}}$ induced by the pushouts in triple graphs $(B +_A C)$ and sets $(\nu_B +_{\nu_A} \nu_C)$.*

Proof. In appendix.

Example. Fig. 5 shows a pushout, where the pushout object D is the result of gluing the constraint triple graphs B and C along the constraint triple graph A, written $B +_A C$. In particular, the resulting constraint has the common nodes A, B and C, whereas graph B adds node D, and graph C adds node E. The formula of D α_D includes the conjunction of the formulas of graphs B and C, and note that $\alpha_D \Rightarrow c'(b(\alpha_A)) \equiv b'(c(\alpha_A))$.

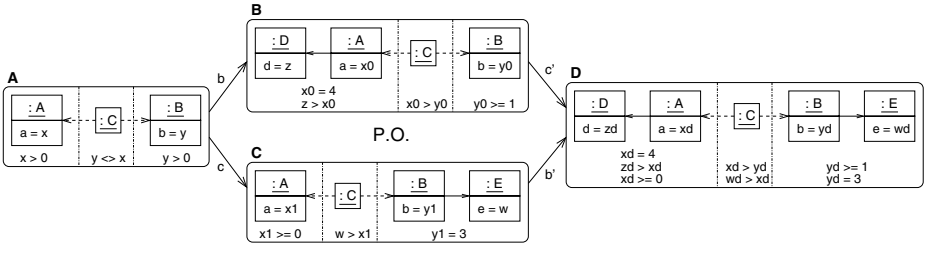


Fig. 5. Pushout example

Sometimes, we have to consider the source or the target parts of a constraint triple graph. The source restriction of a constraint triple graph $CTrG^{\mathcal{A}}$, written $CTrG^{\mathcal{A}}|_S$, is made of the source graph and the source formula, and similarly for the target restriction. Hence, $CTrG^{\mathcal{A}}|_S = (TrG|_S = \langle S, \emptyset, \emptyset \rangle, D^S, \alpha|_{D^S} = \alpha^S)$. The source restriction $CTrG^{\mathcal{A}}|_S$ of a constraint induces a morphism $CTrG^{\mathcal{A}}|_S \hookrightarrow CTrG^{\mathcal{A}}$. Also, given a morphism $q \colon CTrG_1^{\mathcal{A}} \to CTrG_2^{\mathcal{A}}$, we can construct morphism $q_S \colon CTrG_1^{\mathcal{A}}|_S \to CTrG_2^{\mathcal{A}}|_S$ and similarly for the target. This restriction operation will be used later to consider only the source or target models in a constraint, when such constraint is evaluated source-to-target or target-to-source.

An attributed triple graph can be seen as a constraint triple graph whose formula is satisfied by a unique variable assignment, i.e. $\exists_1 \mathfrak{f} : \nu \to \mathcal{A}$ with $\mathcal{A} \models_{\mathfrak{f}} \alpha$. We call such constraints *ground*, and they form the $\mathbf{GroundCTrG}_{\mathcal{A}}$ full subcategory of $\mathbf{CTrG}_{\mathcal{A}}$. We usually depict ground constraints with the attribute

values induced by the formula in the attribute compartments and omit the formula. The equivalence between ground constraints and triple graphs is useful as, from now on, we just need to work with constraint triple graphs.

5 Pattern-Based Model-to-Model Transformation

Now we use the previous concepts to build our M2M specification language. Specifications in our language are made of so called *triple patterns*. These are similar to graph constraints [8], but made of constraint triple graphs instead of graphs. This allows interpreting them both source-to-target and target-to-source.

We consider two kinds of pattern: positive (called P-patterns) and negative (N-patterns). While the former express allowed relations between source and target models, the latter describe forbidden scenarios. A P-pattern has a main constraint (written $P(Q)$), a (possibly empty) positive pre-condition C (written $\overleftarrow{P}(C)$), a set of negative pre-conditions (written $\overleftarrow{N}(C_i)$), and a set of negative post-conditions (written $N(C_j)$). The main constraint of a P-pattern only needs to hold when the positive pre-condition and no negative pre-condition of the pattern hold. If such is the case, then no negative post-condition of the pattern should hold. An N-pattern is a particular case of P-pattern where C and Q are empty, and there is only one negative post-condition $N(C_j)$ which is forbidden to occur (as any negative post-condition). Next definition formalises the syntax of patterns, while Definition 5 describes their semantics.

Definition 4 (Triple Pattern). *Given the injective* $\mathbf{CTrG}_{\mathcal{A}}$*-morphism* $C \xrightarrow{q} Q$ *and the sets of injective* $\mathbf{CTrG}_{\mathcal{A}}$*-morphisms* $N_{Pre} = \{Q \xrightarrow{c_i} C_i\}_{i \in Pre}$, $N_{Post} = \{Q \xrightarrow{c_j} C_j\}_{j \in Post}$ *of negative pre- and post-conditions:*

- $\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j \in Post} N(C_j)$ *is a positive pattern (P-pattern).*
- $N(C_j)$ *is a negative pattern (N-pattern).*

Remark. The notation $\overleftarrow{P}(\cdot)$, $\overleftarrow{N}(\cdot)$, $N(\cdot)$ and $P(\cdot)$ is just syntactic sugar to indicate a positive pre-condition (that we call parameter), a negative pre-condition, a negative post-condition and the main constraint respectively.

The simplest P-pattern is made of a main constraint Q restricted by negative pre- and post-conditions (Pre and $Post$ sets). In this case, Q has to be present in a triple graph (i.e. in a ground constraint) whenever no negative pre-condition C_i is found; and if Q is present, no negative post-condition C_j can be found for the pattern to be satisfied. In this way, while negative pre-conditions express restrictions for the constraint Q to occur, negative post-conditions describe forbidden graphs. If a negative pre-condition is found, it is not mandatory to find Q, but still possible. P-patterns can also have positive parameters, specified with a non-empty C. In such a case, Q has to be found only if C is also found. Finally, an N-pattern is made of one negative post-condition forbidden to occur, and C and Q are empty.

Example. Fig. 6 shows some patterns specifying the consistency between WSDL and EJB models. The P-pattern *P(Definitions-EJBJar)* declares that each `Definitions` object has to be related with an `EJBJar` with same name. This means that the service described in a WSDL document will be handled by a set of related EJBs, bundled in the same *jar* container. The P-pattern *P(Service-SessionBean)* states that each WSDL `Service` is managed by a `Session` bean, made of home and remote interfaces and an implementation class, and with methods to create and initialize the bean. The attribute condition enforces some naming conventions for these. Note that some attribute details (e.g. whether the bean is stateful or stateless) are left open. The pattern has a positive precondition C, which we show in compact notation using *param* tags.

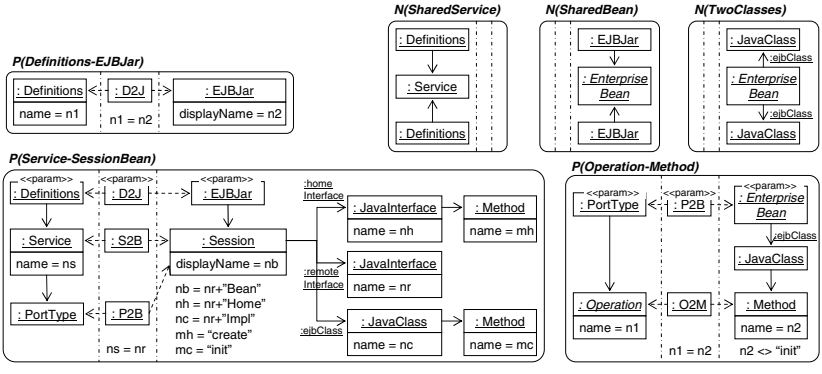


Fig. 6. Some patterns for the WSDL-EJB transformation

The P-pattern *P(Operation-Method)* declares that each `Operation` in a given port type is to be implemented as an EJB `Method` with same name in the bean implementing the port type. The target attribute condition *n2<>"init"* avoids translating the special `init` method created by pattern *P(Service-SessionBean)* back into an operation. In addition, the pattern uses *abstract objects* of types `EnterpriseBean` and `Operation`. This is allowed and, intuitively, it is equivalent to the disjunction of the eight patterns that result from the substitution of the abstract objects by all its concrete subtypes. Thus, the `PortType` may be connected either with a `Session` or with an `Entity`, and the method with any subtype of `Operation`. Finally, three N-patterns forbid `Service`s to belong to two `Definition`s, and an `EnterpriseBean` to belong to two `EJBJar`s and have two `JavaClass`es. Later we will see that in fact these N-patterns can be automatically derived from the meta-models, and also that there is no need to manually specify the parameters in the P-patterns *P(Service-SessionBean)* and *P(Operation-Method)*.

Next, we define pattern satisfaction. Since N-patterns are a special case of P-patterns, a unique definition is enough. Satisfaction is checked on constraint triple graphs, not necessarily ground. This is so because, during a transformation,

the source and target models do not need to be ground. When the transformation finishes a solver can find an attribute assignment satisfying the formulae.

We define forward and backward satisfaction. In the former we check that the main constraint of a pattern is found in all places where the pattern is source-enabled. That is, roughly, in all places where the pre-conditions for enforcing the pattern in a forward transformation hold. The separation between forward and backward satisfaction is useful because if we transform forwards (assuming an initial empty target) we just need to check forward satisfaction. Full satisfaction implies both forward and backward satisfaction and is useful to check if two graphs are actually synchronized. For simplicity, we only enunciate forward satisfaction, see the full definition in [12].

Definition 5 (Satisfaction). *A constraint triple graph $CTrG$ satisfies $CP = [\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j \in Post} N(C_j)]$, written $CTrG \models CP$, iff:*

- *CP is* forward satisfiable*, $CTrG \models_F CP$: $[\forall m^S : P_S \to CTrG$ s.t. $(\forall i \in Pre$ s.t. $N_i^S \ncong P_S$, $\nexists n_i^S : N_i^S \to CTrG$ with $m^S = n_i^S \circ a_i^S)$, $\exists m : Q \to CTrG$ with $m \circ q^S = m^S$, s.t. $\forall j \in Post \nexists n_j : C_j \to CTrG$ with $m = n_j \circ c_j]$, and*
- *CP is* backward satisfiable*, $CTrG \models_B CP$, see [12]*

with $P_x = C +_{C|_x} Q|_x$, $N_i^x = C +_{C|_x} C_i|_x$ and $N_i^x \xleftarrow{a_i^x} P_x \xrightarrow{q^x} Q$ $(x \in \{S,T\})$, see left of Fig. 7. $C +_{C|_x} Q|_x$ is the pushout object of C and $Q|_x$ through $C|_x$.

Remark. We use the notation $A \cong B$ to denote that A and B are isomorphic, and $A \ncong B$ to denote that A and B are not isomorphic.

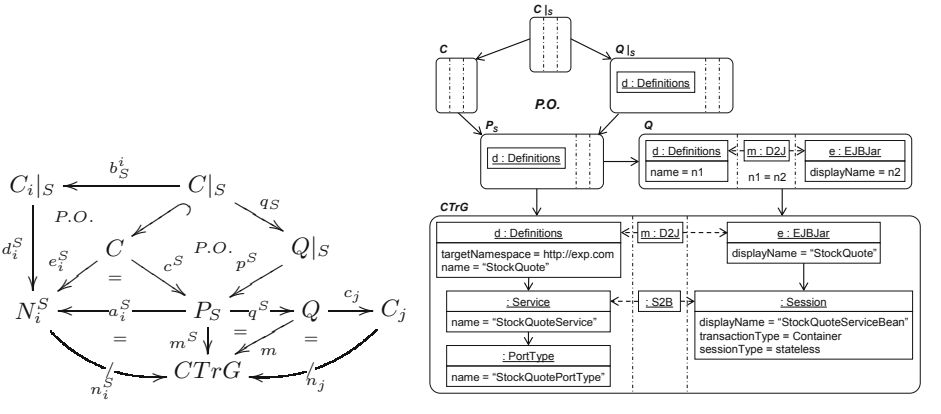


Fig. 7. Forward satisfaction (left). Example (right).

Example. The right of Fig. 7 shows an example of forward satisfaction of pattern *P(Definitions-EJBJar)* by a ground constraint triple graph *CTrG*. There is one occurrence of the source restriction of the pattern in *CTrG*, which can be extended to the whole pattern. In addition, *CTrG* also backward-satisfies the

pattern and hence it satisfies it. Note however that *CTrG* does not forward-satisfy pattern *P(Service-SessionBean)* as the session bean does not define the required java classes and interfaces. The satisfaction checking of a pattern with abstract objects is the same as that of a pattern without them, thus enabling the usual allowed substitution of abstract types by concrete ones.

We can distinguish several kinds of pattern satisfaction. In *trivial satisfaction*, a pattern is satisfied because no morphism m^S exists (i.e. there is no occurrence of the source restriction of the pattern). This is for example the case of pattern *P(Operation-Method)* in the constraint *CTrG* of Fig. 7, as there is no `Operation` object in the source of the constraint. In *vacuous satisfaction*, a pattern is satisfied because m^S exists but some of its negative pre-conditions are also found. In this case, the main constraint Q of the pattern is not demanded to occur in *CTrG*. Finally, in *positive satisfaction*, m^S and m exist and the negative pre- and post-conditions are not found. All these three cases are handled by Definition 5.

One M2M specification is a conjunction of patterns, and hence a constraint triple graph satisfies a specification if it satisfies all its patterns.

5.1 Considering the Meta-model Integrity Constraints

A transformation specification cannot be oblivious of the meta-model integrity constraints. The simplest ones are the maximum cardinality constraints in association ends. These induce N-patterns that in this paper we automatically derive and include in the transformation specification. This is useful to prevent the operational mechanisms from generating syntactically incorrect models, as N-patterns will be transformed into post-conditions of the operational rules.

The generation procedure is simple: if a class `A` is restricted to be connected to a maximum of `j` objects of type `B`, then we build an N-pattern made of an `A` object connected to `j+1 B` objects. As an example, Fig. 6 showed three N-patterns that were derived from the WSDL and EJB meta-model constraints.

Note that additional (but restricted) forms of OCL could also be transformed, and here we can benefit from previous works on translating OCL into graph constraints [31]. Interestingly, once the meta-model constraints are expressed in the form of patterns, we can analyse their consistency with the rest of the specification. For example, if we find a morphism from some of the generated N-patterns to an existing P-pattern, then we can conclude that the transformation is incorrect, as it could try to create models violating the cardinality constraints. We plan to develop further static analysis techniques, similar to those of [22].

6 Generation of Operational Mechanisms

This section describes the synthesis of TGG operational rules implementing forward and backward transformations from pattern-based specifications. In forward transformation, we start with an initial constraint triple graph with correspondence and target empty, and the other way round for backward transformation. Moreover, we also assume that the source or target initial models

do not violate any N-pattern of the specification. Recall that some of these N-patterns are derived from the maximum association cardinality constraints in meta-models, and hence it is reasonable to assume syntactically correct starting models.

The synthesis process derives one rule from each P-pattern, made of triple constraints in its LHS and RHS. In particular, $P_S = C +_{C|_S} Q|_S$ is taken as the LHS for the forward rule, and the main constraint Q as the RHS. As an example, Fig. 8 shows the LHS and RHS of the forward rule derived from pattern *P(Definitions-EJBJar)*.

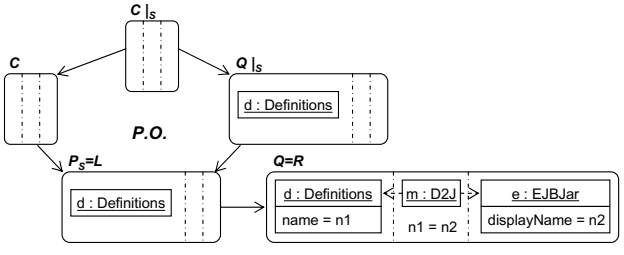


Fig. 8. Forward rule generation example

If a rule creates objects having a type with defined subtypes, we generate a set of rules resulting from substituting the type by all its concrete subtypes in the graph created by the rule, i.e. the nodes in $RHS \setminus LHS$. This substitution is not necessary in the elements of the LHS as they are not created, and it is not done in the NACs either in order to obtain the expected behaviour of disjunction. Using an optimization similar to [3], one could also work directly with abstract rules, but we would have to modify the notion of morphism and it is left for future work.

The negative pre- and post-conditions of a P-pattern are used as negative pre- and post-conditions of the associated rule(s). All N-patterns are converted into negative post-conditions of the rule(s), using the well-known procedure to convert graph constraints into rule's post-conditions [8]. Finally, additional NACs are added to ensure termination. For simplicity, we only show the generation of the forward rules, the backward rules are generated analogously [12].

Definition 6 (Derived Forward Rule). *Given specification SP and $P = [\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j\in Post} N(C_j)] \in SP$, the set of forward rules $\overrightarrow{r_P} = \{((L = C +_{C|_S} Q^n|_S \overset{r^n}{\rightarrow} R^n = Q^n), pre^n(P), post^n(P))\}_{n\in Conc(P)}$ is derived, where $\{L \overset{r^n}{\rightarrow} Q^n\}_{n\in Conc(P)}$ is the set of rules $L \rightarrow Q^n$ resulting from all valid substitutions of types by concrete subtypes in nodes belonging to $V^Q \setminus r(V^L)$. The set $pre^n(P)$ of NACs is defined as the union of the following two sets:*

- $NAC(P) = \{L \xrightarrow{a_i^S} N_i^S | L \ncong N_i^S\}_{i \in Pre}$ *is the set of NACs derived from* P*'s negative pre-conditions, with* $N_i^S \cong C_i|_S +_{C|_S} C$*. See the left of Fig. 7, where* P_S *is* L *in this definition.*
- $TNAC^n(P) = \{L \xrightarrow{m_k} T_k\}$ *is the set of NACs ensuring termination, where* T_k *is built by making* m_k *injective and jointly surjective with* $Q^n \xrightarrow{f} T_k$*, s.t. the diagram shown below commutes.*

$$\begin{array}{ccc} Q^n|_S & \rightarrow & Q^n \\ \downarrow & = & \downarrow f \\ L & \longrightarrow & T_k \end{array}$$

and the set $post^n(P)$ *is defined as the union of the following two sets of negative post-conditions:*

- $POST^n(P) = \{m_j \colon R^n \rightarrow C_j\}_{j \in Post}$ *is the set of rule's negative post-conditions, derived from the set of* P*'s post-conditions.*
- $NPAT^n(P) = \{R^n \rightarrow D | [N(C_k)] \in SP,\ R^n \rightarrow D \leftarrow C_k$ *is jointly surjective, and* $(R^n \setminus L) \cap C_k \neq \emptyset\}$ *is the set of negative post-conditions derived from each N-pattern* $N(C_k) \in SP$*.*

Remarks. In the previous definition, we have used function *Conc(P)*, which given a pattern P, calculates the set of all valid node type substitutions $\{Q^n\}$ of its main constraint Q. Slightly abusing the notation, we have used $Conc(P)$ as an index set.

The set $NPAT^n(P)$ contains the negative post-conditions derived from the N-patterns of the specification. This is done by relating each N-pattern with the rule's RHS in each possible way. Moreover, the requirement that $(R^n \setminus L) \cap C_k \neq \emptyset$ reduces the size of $NPAT^n(P)$, because we only need to consider possible violations of N-patterns due to created elements by the RHS, as we start with an empty target model, and the source already satisfies all N-patterns.

Example. The upper row of Fig. 9 shows the operational forward rule generated from pattern *P(Service-SessionBean)*, which does not contain abstract objects. There are three NACs for termination, `TNAC1`, `TNAC2` and `TNAC3`, the former equal to R. `TNAC3` is not shown in the figure, but is like `TNAC1` with an additional node of type `D2J` in the correspondence graph, connecting nodes `d` and `e`. Note that we do not do any algebraic manipulation of formulae to generate the rule, hence demonstrating the advantages of using constraint triple graphs in our approach. The figure also shows a direct derivation where both G and H are ground constraints. Constraint H is obtained by a pushout, and hence according to Prop. 1 is calculated by a pushout on triple graphs and the conjunction of the formulae of R and G. When the transformation ends, a constraint solver can be used to resolve attribute values. We will present further generated rules in Section 8.

According to [21], the generated rules are terminating and, in absence of N-patterns, correct: they produce only valid models of the specification. However, the rules are not complete: not all models satisfying the specification can be

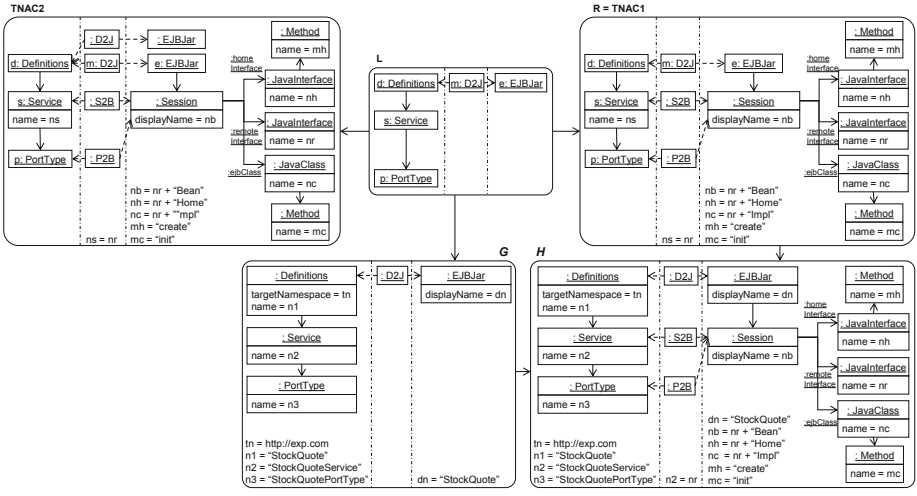


Fig. 9. Generated forward rule and derivation

produced. For example, assume we have a starting model with two `Definitions` objects with same name. Then, the synthesized forward rules are able to generate the model to the left of Fig. 10, but not the one to the right of the same figure, which also satisfies the specification. The model to the right would be generated if we could synthesize rules reusing elements created by previous applications of rules. Next subsection describes a method, called *parameterization*, that ensures completeness of the rules generated from a specification without N-patterns (and therefore it makes possible to find both solutions in the figure). The main idea is to generate additional patterns with increasingly bigger parameters, which enables the generated rules to reuse previously created elements.

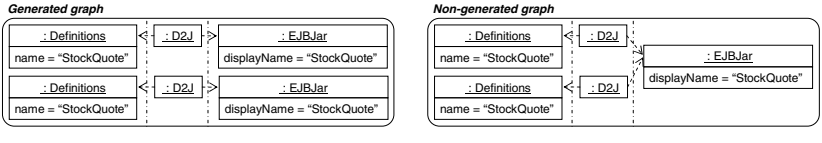


Fig. 10. Reachable (left) and unreachable (right) models for the specification without parameterization

Please note that the resulting constraint of a forward transformation forward-satisfies the specification, but does not necessarily backward-satisfies it. This is also noticed in QVT-R [23], where check-only transformations are directed as well (either forwards or backwards). Thus, the result of an *enforcing* forward transformation does not necessarily satisfy the same transformation when executed backwards in mode check-only, and vice versa.

If a specification contains arbitrary N-patterns, these are added as negative post-conditions for the rules, preventing the occurrence of N-patterns in the model. However, they may forbid applying any rule before a valid model is found, thus producing graphs that may not satisfy all P-patterns. In this case, some terminal graphs – to which no further rule can be applied – may not be models of the specification. Note however that if the specification admits solutions, our operational mechanisms are still able to find all of them, but in this case not all terminal models with respect to the grammar satisfy the specification.

6.1 Parameterization and Heuristics for Rule Derivation

In order to obtain completeness, we apply an operation called *parameterization* to every P-pattern in the specification. In this way, the resulting rules are able to generate all possible models of the specification [12,21]. The *parameterization* operation takes a P-pattern and generates additional ones, with all possible positive pre-conditions "bigger" than the original pre-condition, and "smaller" than the main constraint Q. This allows the rules generated from the patterns to reuse already created elements.

Definition 7 (Parameterization). *Given* $T = \bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j \in Post} N(C_j)$*, its parameterization is* $Par(T) = \{\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C') \Rightarrow P(Q) \bigwedge_{j \in Post} N(C_j) | C \overset{i_1}{\hookrightarrow} C' \overset{i_2}{\hookrightarrow} Q, C \ncong C', C' \ncong Q\}$.

Remark. The formula $\alpha_{C'}$ can be taken as the conjunction of α_C for the variables already present in ν_C, and α_Q for the variables not in ν_C (i.e. in $\nu'_C \backslash i_1(\nu_C)$). Formally, $\alpha_{C'} = \alpha_C \wedge \alpha_Q|_{i_2(\nu_{C'} \backslash i_1(\nu_C))}$ (assuming no renaming of variables).

Example. Fig. 11 shows some of the parameters generated by parameterization for a pattern like *P(Operation-Method)* in Fig. 6 but without parameters. Parameterization generates 123 patterns in total. The pattern with parameter $\overleftarrow{P}(1)$ is enforced when the port is already mapped to an EJB with a Java class, and in forward transformation avoids generating a rule that creates a bean and a Java class with arbitrary names. Parameter $\overleftarrow{P}(3)$ reuses an operation with the same name as the method, and in backward transformation allows generating just one operation from a number of methods with the same name but different number of parameters. However, $\overleftarrow{P}(2)$ is potentially harmful as it may lead to reusing a method that already belongs to a different bean, and thus to an incorrect model. Note however that this is not possible as an N-pattern generated from the maximum cardinality constraints of the meta-model forbids methods to belong to two different `JavaClass`es. This shows that including the cardinality constraints of the meta-models as N-patterns in the transformations allows controlling the level (and correctness) of reuse.

As the example shows, parameterization generates an exponential number of patterns with increasingly bigger parameters, resulting in an exponential number of rules. However one does not need to generate these rules beforehand, but they can be synthesized "on the fly". Moreover, some of these forward rules generated

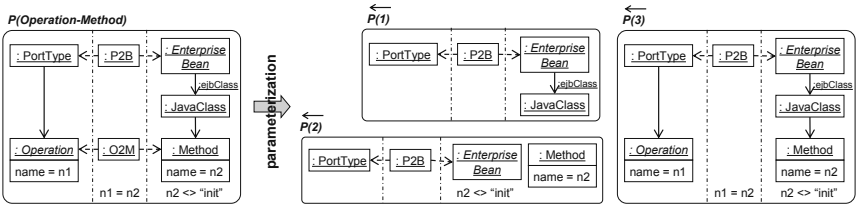


Fig. 11. Parameterization example

from the parameterized pattern will actually be equal, namely, those generated from parameters with same target and correspondence graph. Although parameterization ensures completeness, we hardly use it in practice due to the high number of generated rules, and we prefer using heuristics to control the level of reuse. However, as previously stated, generating fewer patterns can make the rules unable to find certain models of the specifications (those "too small").

In order to reduce the number of rules, we propose two heuristics. The first one is used to derive only those parameters that avoid creation of elements with unconstrained attribute values. The objective is to avoid synthesizing rules that create elements whose attributes can take any value. Instead, we prefer that these elements are generated by some other rule that assigns them a value, if it exists. Note that some transformations may not provide a unique value for each attribute thus being "loose".

Heuristic 1. *Given a pattern P, replace it by a new pattern that has as parameter all elements with some attribute not constrained by the formula in P but constrained by some other pattern, as well as the mappings and edges between these elements. We do not apply the heuristic if the obtained parameter is equal to Q.*

Example. In the pattern in Fig. 11, the heuristic generates just one pattern with parameter $\overleftarrow{P}(1)$. Thus, the generated forward rules do not create beans or classes with arbitrary names. Note that the heuristic replaces the original pattern with the generated one. This example shows that there is no need to set this parameter explicitly a priori as we did in the initial specification of Fig. 6.

In Fig. 12 we present to the left an extended version of the pattern *P(Definitions-EJBJar)* which maps WSDL definitions to a jar but also to a package. In the backward direction, a definitions object will be created for each package and its container jar (this is known because the package contains an interface that belongs to a bean inside the jar), and we use the name of the package to give value to the `targetNamespace` attribute. However, in the forward direction we want to avoid the creation of beans with undefined name, therefore we apply the presented heuristic and obtain the pattern to the right, where the positive pre-condition is annotated with the key $\langle\langle param \rangle\rangle$ and highlighted. In addition, we need to ensure that there are no two `Definitions` with same `name` and `targetNamespace`, otherwise the operational mechanisms would

create different `Definition` objects for each `JavaInterface` inside a package. This can be done using one N-pattern, or as we will see later, using the CBE semantics.

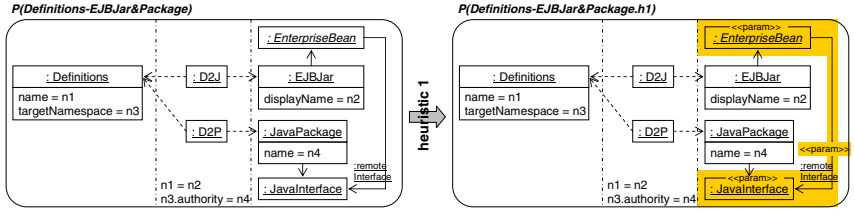


Fig. 12. Applying heuristic 1 to a pattern

The next heuristic generates only those parameters that avoid duplicating a graph S_1, forbidden by some N-pattern of the form $N(S_1 +_U S_1)$. This ensures the generation of rules producing valid models for the class of specifications with N-patterns of this form (called FIP in [4]), and which include the N-patterns generated by the maximum cardinality constraints in meta-models. The way to proceed is to apply heuristic 2 to each P- and N-pattern of the form $N(S_1 +_U S_1)$, and repeat the procedure with the resulting patterns until no more different patterns are generated.

Heuristic 2. *Given a P-pattern* $[\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q)] \in SP$*, if there is an N-pattern* $[N(S)] \in SP$ *with* $S \cong S_1 +_U S_1$*, and* $\exists s\colon S_1 \to Q, u\colon U \to C$ *s.t.* $s \circ u_1 = q \circ u$ *(see left of Fig. 13), and* $\nexists s'\colon S_1 \to C$ *all injective s.t.* $q \circ s' = s$*, then we generate additional patterns with parameters all* C'_j *s.t.* q_1 *and* q_s *in Fig. 13 are jointly surjective, and the induced* $C'_j \to Q$ *is injective.*

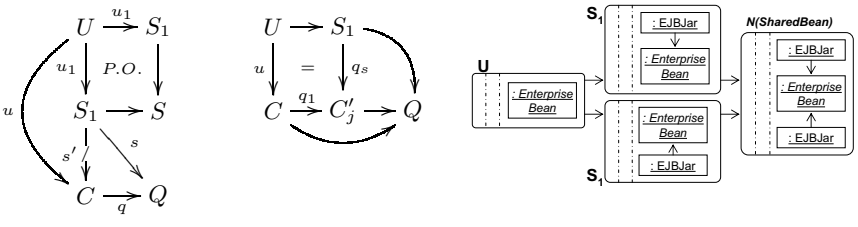


Fig. 13. Condition for heuristic 2 and generated parameters (left and center). Decomposition of N-pattern (right).

The rationale of this heuristic is that if a P-pattern has a parameter C that contains U but not S_1, and its main constraint Q contains S_1, then applying the pattern creates a new structure S_1 glued to an existing occurrence of U. This heuristic enlarges the parameter to include S_1 and thus avoid its publication. The way to proceed is to apply the heuristic for each P- and N-pattern of the

form $N(S_1 +_U S_1)$, and repeat the procedure with the resulting patterns until no more different patterns are generated.

Example. The right of Fig. 13 shows that N-pattern *N(SharedBean)* satisfies the conditions demanded by heuristic 2. The pattern forbids an `EnterpriseBean` to belong to two `EJBJars`. Fig. 14 shows the application of heuristic 2 to the P-pattern *P(Definitions-EJBJar&Package.h1)* previously obtained by heuristic 1, and to the N-pattern *N(SharedBean)* decomposed in Fig. 13. The generated parameter C'_1 includes the `EJBJar` so that it is not created in forward transformation. In this way, it avoids the creation of the model fragment forbidden by the N-pattern. Note that the initial pattern with parameter C is also kept in the specification.

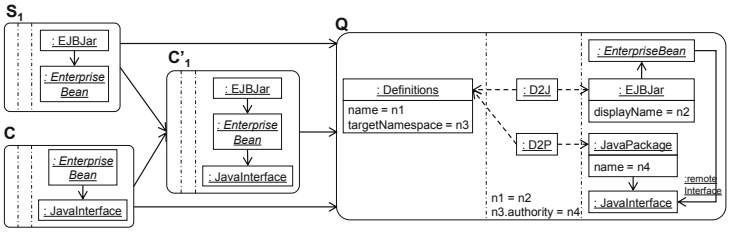


Fig. 14. Applying heuristic 2

7 Check-Before-Enforce Semantics

Even though the presented heuristics help controlling the level of reuse, in an M2M transformation it is useful to control whether an element has to be created in the generated domain or whether it already exists and can be reused. This avoids creating duplicated objects. This control mechanism has been incorporated to approaches like QVT and is called Check-Before-Enforce (CBE) semantics. In this section we incorporate it to our framework.

The idea is to generate N-patterns forbidding two objects of the same type with the same attribute values. Then, our *Heuristic 2* takes each P-pattern in the specification and generates new ones with appropriate parameters reusing the objects whenever possible.

Example. The left of Fig. 15 shows the N-pattern that the CBE semantics generates for class `Definitions`, which forbids two `Definitions` objects with same attribute values in the WSDL model. The center of the same figure presents the pattern generated by heuristic 2 from pattern *P(Definitions-EJBJar&Package.h1)* shown in Fig. 12 due to the newly introduced N-pattern. The new pattern adds a `Definitions` object to the previous parameter.

To allow for a better control of reuse, and to permit the specification of when two objects are to be considered equal, QVT includes the concept of *Key*. Keys allow us, for example, to neglect certain attributes when comparing if two

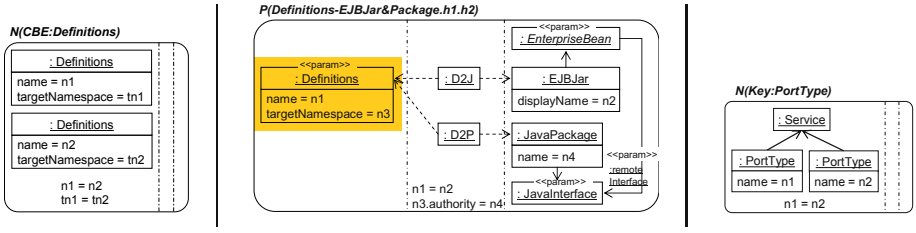


Fig. 15. N-pattern generated by CBE semantics (left). Pattern generated by heuristic 2 (center). N-pattern generated from the key of `PortType`.

objects are the same, or include further connected objects in the comparison. Again, such a concept can be easily incorporated into our framework by an appropriate generation of N-patterns. For instance, we can set that the key for `PortType`s is their name and owner service. This would be specified in QVT as `Key PortType{name, Service}`, from which our procedure generates the N-pattern to the right of Fig. 15.

8 Specification Process and Back to the Case Study

Fig. 16 summarises the steps needed to engineer a pattern-based transformation specification and obtain the operational mechanisms. The process is shown as a SPEM model [26], similar to an activity diagram, where activities are numbered in dots and represented as arrow-like icons. The model distinguishes the level at which the activity is performed (language, specification or operational) and who performs it (language engineer, transformation engineer or automated process).

First, the language engineer designs the source and target meta-models or reuse them if already available (*step 1*). Next, the transformation engineer designs the allowed traces between the elements in the source and target languages, obtaining a meta-model triple as a result (*step 2*). Once this is available, several activities can start in parallel. On the one hand, the engineer builds the transformation specification (*step 3a*) and sets the keys (*step 3b1*). On the other hand, our automatic mechanisms generate the N-patterns derived from the meta-model constraints (*step 3c*), as well as those for the CBE semantics and keys (*step 3b2*). Then, we apply the heuristics to the transformation specification and N-patterns synthesized by the previous activities (*step 4*). This results in an enriched specification that is used to generate the TGG operational rules, once the transformation direction is chosen (*step 5*).

If we apply this engineering process to our case study, the first step is to build the WSDL and EJB meta-models, which were shown in Figs. 1 and 2. The trace meta-model defines four types of nodes: (i) `D2J` connecting `Definitions` and `EJBJar` objects, (ii) `S2B` connecting `Service` and `Session` objects, (iii) `P2B` connecting `PortType` and `Session` objects, and (iv) `O2M` connecting `Operation` and `Method` objects.

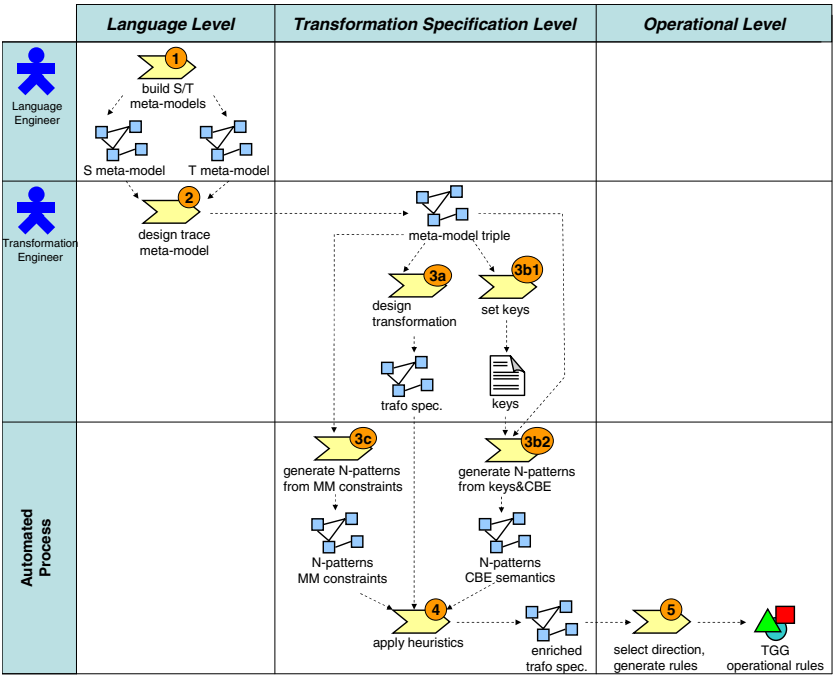


Fig. 16. Our transformation engineering process

Next, we have to design the transformation specification. This is made of the three P-patterns shown in Fig. 6: *P(Definitions-EJBJar)*, *P(Service-SessionBean)* and *P(Operation-Method)*, all of them without parameters. For simplicity we do not consider Java packages, and thus omit pattern *P(Definitions-EJBJar&Package)* shown in Fig. 12.

Meanwhile, *step 3c* generates one N-pattern for each association end in the meta-model with bounded upper cardinality. Three of these N-patterns were shown in Fig. 6. In its turn, *step 3b2* generates additional N-patterns due to the CBE semantics and according to the specified keys. This results in one N-pattern for each class in the meta-models. In case we chose the direction of the transformation first, it would be enough to generate N-patterns from one of the meta-models: the target in forward transformations and the source in backwards. Fig. 15 showed some of the N-patterns generated due to CBE semantics. Then, applying the heuristic 1 replaces some P-patterns by others with parameters, and applying the heuristic 2 adds new patterns to the specification.

Finally, we choose the operational scenario to be solved and generate the TGG operational rules. Two of the generated forward rules are shown in Fig. 17. The rule to the left is generated from the *P(Definitions-EJBJar)* pattern and creates an `EJBJar` object for each `Definitions` object in the WSDL model. The rule has one termination NAC equal to the RHS, and two post-conditions coming from the N-pattern *N(SharedBean)* that was derived by a meta-model cardinality constraint.

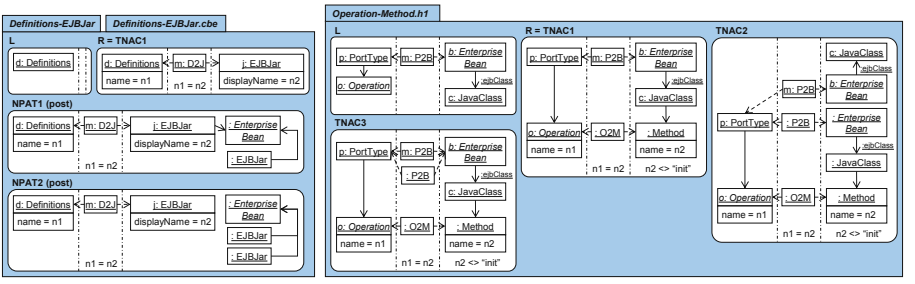


Fig. 17. Some of the generated forward rules for the case study

The right of the same figure shows the rule generated from pattern *P(Operation-Method.h1)*, pattern that replaced pattern *P(Operation-Method)* after applying the heuristic 1. The rule creates one `Method` for each `Operation` in a `PortType`. Note that objects `o` and `b` have abstract type. The rule has three termination NACs, as well as several negative post-conditions that are omitted for simplicity.

The generated forward rules can be applied to WSDL models in order to obtain the EJB model. Fig 18 shows one example where we start with a WSDL containing one `Service` owning a `PortType` with two `Operations`. After applying the four rules shown in the figure, we obtain the constraint triple graph to the right, to which no more rules can be applied. Note that in this final model not all attributes are constrained, for example the `transactionType` and the `sessionType` of the `Session` object (these attributes are not considered by the transformation specification). Hence, a constraint solver could give arbitrary values to these attributes, or the user could be asked to give one. Also, for this starting model the transformation is confluent (in the sense that we obtain a

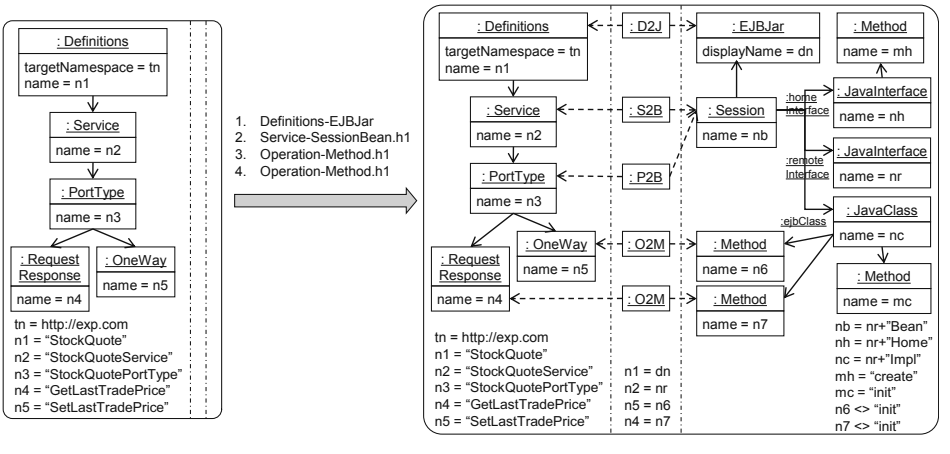


Fig. 18. Example model transformation

unique constraint triple graph, from which we can however derive several models by assigning different attribute values), but this is not necessarily so for other models, as already noted in [12,21]. This is actually a good behaviour, as one obtains all terminal models that satisfy the transformation specification.

9 Conclusions and Future Work

In this paper we have incorporated the CBE semantics and keys concepts of QVT-R into our pattern-based M2M framework in order to control object reuse in M2M transformations. This is achieved by adding N-patterns to the specification so that they forbid the existence of two objects that are considered equal, thus making the operational mechanism to reuse such objects whenever possible. We have also shown that the meta-model inheritance hierarchy and the integrity constraints have to be considered by the transformation specification. In particular, we have discussed how to generate N-patterns from the maximum cardinality constraints in associations, as well as how to handle abstract objects in patterns. Finally, we have illustrated these concepts with a transformation between WSDL and EJB models.

There are many open lines for further research. For example, one could consider the benefits of adding relations between the nodes in the correspondence graph instead of having a discrete graph there. These have been exploited in [9] for implementing incremental transformations, but note that our theory demands graph morphisms between the correspondence and the other two graphs, hence posing some restrictions. We are also starting to investigate more complex operational scenarios, like incremental transformations and model synchronization. On the theoretical side, it is worth investigating analysis methods for specifications, as well as simplifications of the current formalism. For example, in our experience, it seems possible to get rid of parameters in the initial specification, and express the restrictions with N-patterns so that one ends up with equivalent specifications. However, this is still an open question. We would also like to explore higher-level means of specifications, by (i) omitting the correspondence graph at the specification level (and automatically generating the traces at the operational level, as in [10,14]), and (ii) making possible the specification of pattern dependencies and parameter passing, similar to when or where clauses in QVT. These two steps would allow us to express the semantics of QVT-R with our framework. We also plan to perform a detailed study of the expressivity of different mechanisms for reuse of other bidirectional languages, like TGGs and QVT-R, by using realistic examples. Finally, we are also investigating other languages for the operational mechanisms, like Coloured Petri Nets, in the style of [5].

Acknowledgements. Work partially supported by the Spanish Ministry of Science and Innovation, with projects METEORIC (TIN2008-02081) and FORMALISM (TIN2007-66523), and the R&D program of the Community of Madrid (S2009/TIC-1650, project "e-Madrid"). Moreover, part of this work was done during a post-doctoral stay of the first author at the University of York, and sabbatical leaves of the second and third authors to the University of York and TU

Berlin respectively, all with financial support from the Spanish Ministry of Science and Innovation (grant refs. JC2009-00015, PR2009-0019 and PR2008-0185).

References

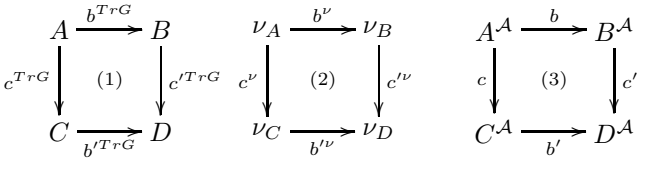
1. Akehurst, D.H., Kent, S.: A relational approach to defining transformations in a metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002)
2. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. ENTCS 72(3) (2003)
3. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. TCS 376(3), 139–163 (2007)
4. de Lara, J., Guerra, E.: Pattern-based model-to-model transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 426–441. Springer, Heidelberg (2008)
5. de Lara, J., Guerra, E.: Formal support for QVT-Relations with coloured Petri nets. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 256–270. Springer, Heidelberg (2009)
6. Dehayni, M., Féraud, L.: An approach of model transformation based on attribute grammars. In: Konstantas, D., Léonard, M., Pigneur, Y., Patel, S. (eds.) OOIS 2003. LNCS, vol. 2817, pp. 412–423. Springer, Heidelberg (2003)
7. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer, Heidelberg (2006)
9. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1), 21–43 (2009)
10. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: Implementing Query/View/Transformation with Triple Graph Grammars. Software and Systems Modeling 9(1), 21–46 (2010)
11. Guerra, E., de Lara, J.: Event-driven grammars: Relating abstract and concrete levels of visual languages. Software and Systems Modeling, special section on ICGT 2004 6(3), 317–347 (2007)
12. Guerra, E., de Lara, J., Orejas, F.: Pattern-based model-to-model transformation: Handling attribute conditions. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 83–99. Springer, Heidelberg (2009)
13. IBM WebSphere, http://www-01.ibm.com/software/websphere/
14. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report TR-RI-07-284, Paderborn University (2007)
15. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended triple graph grammars with compatible graph translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 141–174. Springer, Heidelberg (2010)
16. Königs, A., Schürr, A.: Tool integration with triple graph grammars - a survey. ENTCS 148(1), 113–150 (2006)
17. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled. Addison-Wesley Object Technology Series (2004)
18. MTF. Model Transformation Framework, http://www.alphaworks.ibm.com/tech/mtf

19. OMG: Metamodel and UML profile for Java and EJB specification (2004), http://www.omg.org/cgi-bin/doc?formal/04-02-02.pdf
20. Oracle containers for J2EE, http://www.oracle.com/technology/tech/java/oc4j
21. Orejas, F., Guerra, E., de Lara, J., Ehrig, H.: Correctness, completeness and termination of pattern-based model-to-model transformation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 383–397. Springer, Heidelberg (2009)
22. Orejas, F., Wirsing, M.: On the specification and verification of model transformations. In: Palsberg, J. (ed.) Semantics and Algebraic Specification. LNCS, vol. 5700, pp. 140–161. Springer, Heidelberg (2009)
23. QVT (2008), http://www.omg.org/spec/QVT/1.0/PDF/
24. Roman, E., Sriganesh, R.P., Brose, G.: Mastering Enterprise JavaBeans, 3rd edn. Wiley, Chichester (2004)
25. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
26. SPEM (2008), http://www.omg.org/cgi-bin/doc?formal/08-04-01.pdf
27. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
28. Völter, M., Stahl, T.: Model-driven software development. Wiley, Chichester (2006)
29. W3C: WSDL v1.1. specification (2001), http://www.w3.org/TR/wsdl
30. W3C: SOAP v1.2. specification (2007), http://www.w3.org/TR/soap
31. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. ENTCS 211, 159–170 (2008)

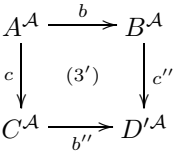
Appendix

Proof of Proposition 1

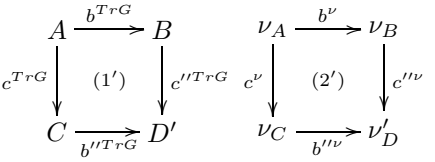
Proof. We have to prove that if diagrams (1) and (2) are pushouts then diagram (3) is also a pushout, where $D^{\mathcal{A}} = (D, \nu_D, c'(\alpha_B) \wedge b'(\alpha_C))$.



First, it may be noted that diagram (3) is indeed a diagram in $\mathbf{CTrG}_{\mathcal{A}}$, since $(c'(\alpha_B) \wedge b'(\alpha_C)) \Rightarrow c'(\alpha_B)$ and $(c'(\alpha_B) \wedge b'(\alpha_C)) \Rightarrow b'(\alpha_C)$ are tautologies, which means that b' and c' are indeed morphisms in $\mathbf{CTrG}_{\mathcal{A}}$. Moreover, we know that if diagram (3') commutes:



then also diagrams (1') and (2') commute:



which means that there are unique morphisms $e^{TrG} : D \to D'$ and $e^{\nu} : \nu_D \to \nu_{D'}$ satisfying $e^{TrG} \circ b'^{TrG} = b''^{TrG}$, $e^{TrG} \circ c'^{TrG} = c''^{TrG}$, $e^{\nu} \circ b'^{\nu} = b''^{\nu}$, and $e^{\nu} \circ c'^{\nu} = c''^{\nu}$. But this means that $e : (D, \nu_D, c'(\alpha_B) \wedge b'(\alpha_C)) \to (D', \nu_{D'}, \alpha_{D'})$ is a morphism, since if $\mathcal{A} \models \alpha_{D'} \Rightarrow c''(\alpha_B)$ and $\mathcal{A} \models \alpha_{D'} \Rightarrow b''(\alpha_C)$ then $\mathcal{A} \models (\alpha_{D'} \Rightarrow (c''(\alpha_D) \wedge b''(\alpha_C)))$. But we know that $c''(\alpha_B) \wedge b''(\alpha_C) = (e \circ c')(\alpha_B) \wedge (e \circ b')(\alpha_C) = e(c'(\alpha_B) \wedge b'(\alpha_C)))$ and this means that $\mathcal{A} \models (\alpha_{D'}) \Rightarrow e(c'(\alpha_B) \wedge b'(\alpha_C))$. Finally, if $e' : (D, \nu_D, c'(\alpha_B) \wedge b'(\alpha_C)) \to (D', \nu_{D'}, \alpha_{D'})$ is a morphism satisfying that $e' \circ b' = b''$ and $e' \circ c' = c''$ then, by the uniqueness of e^{TrG} and e^{ν}, we have that $e = e'$.

Lessons Learned from Building a Graph Transformation System

Gabor Karsai

Institute for Software-Integrated Systems, Vanderbilt University,
Nashville, TN 37205, USA
gabor.karsai@vanderbilt.edu

Abstract. Model-driven software development is a language- and transformation-based paradigm, where the various development tasks of engineers are cast in this framework. During the past decade we have developed, evolved, and applied in practical projects a manifestation of this principle through a suite of tools we call the Model-Integrated Computing suite. Graph transformations are fundamental to this environment and tools for constructing model translators, for the specification of the semantics of languages, for the evolution of modeling languages, models, and their transformations have been built. Designing and building these tools have taught us interesting lessons about graph transformation techniques, language engineering, scalability and abstractions, pragmatic semantics, verification, and evolutionary changes in tools and designs. In the paper we briefly summarize the techniques and tools we have developed and used, and highlight our experience in constructing and using them.

Keywords: model-driven development, integrated development environments, graph transformations, model transformations.

1 Introduction

Model-driven software development is a language- and transformation-based paradigm, where the various development tasks of engineers are cast in this framework [35]. Models are used in every stage of the software product's lifecycle and the model-oriented thinking about the product permeates every aspect of the software engineer's work. Models are used to capture requirements and designs, assist in the implementation, verification, testing, deployment, maintenance, and evolution.

As there is no single language or tool that solves all these problems in software production no single modeling language or modeling tool can solve them either - hence a multitude of models is needed. Models are the artifacts of software production, and there are dependencies among these models: some models are closely related to each other (e.g. design models to requirement models), while some models (and other, non-model artifacts) are automatically generated from models. Two examples for the latter include 'analysis models' that are suitable for verification in some automated analysis tool (e.g. SMV) and executable code (e.g. in C); both of them are derived from the same source model (e.g. UML State Machines).

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 202–223, 2010.

For a model-driven development process model transformations are essential: these transformations connect the various models and other artifacts, and they need to be executed frequently by the developers, or by some automated toolchain. Hence, constructing model transformation tools is of great importance for the builders of development toolchains. The model transformations have to be correct, reliable, robust, and provide high performance; otherwise the productivity of developers is reduced.

The advantages of using domain-specific approaches to software development and modeling are well recognized [1]. Using domain-specific modeling languages necessitates the development of custom, domain-specific model transformations – that are subject to the same quality requirements as any other transformations in a toolchain.

In the past 15+ years, our team has created and evolved a tool-suite for model-driven software development that we call 'Model-Integrated Computing' (MIC) suite [24]. The toolsuite is special in the sense that emphasizes (and encourages) the use of domain-specific models (and thus modeling languages), as opposed to focusing on a single general purpose approach (like UML). Hence model transformations (and especially domain-specific model transformations) play an essential role in the MIC suite. Another specialty of the suite is that it is a 'meta-toolsuite' as it allows defining and constructing domain-specific toolchains with dedicated domain-specific modeling languages.

The development of the MIC toolsuite involved creating the technology for all aspects of a domain-specific model-driven toolchain, including language definition (including concrete and abstract syntax, as well a semantics), model editing, specifying model transformations, the verification of models and model transformations, code generation, the evolution of models and model transformations. In this paper we focus on the model transformation aspects of the toolsuite and present what interesting lessons have been learned about graph transformation techniques, language engineering, scalability and abstractions, pragmatic semantics, verification, and evolutionary changes in tools and designs. In the text we will *indicate important lessons using the mark* [L].

The paper is organized as follows. First, we discuss the fundamental concepts related domain-specific modeling languages. Next, the main ideas used in model transformations are introduced; followed by the discussion on four selected problem domains: efficiency, practical use of transformations, verification of transformations, and the role of transformations in evolution and adaptation. The paper concludes with a summary and topics for further research.

2 Foundations: Metamodels

The first problem in constructing a domain-specific model-driven toolchain one faces is the specification and definition of domain-specific modeling languages (DSML) [24]. Formally, a DSML L is a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_S, and M_C):

$$L = < C, A, S, M_S, M_C >$$

The *concrete syntax* (C) defines the specific (textual or graphical) notation used to express models, which may be graphical, textual or mixed. The *abstract syntax* (A)

defines the *concepts, relationships,* and *well-formedness constraints* available in the language. Thus, the abstract syntax determines all the (syntactically) correct "sentences" (in our case: models) that can be built. It is important to note that the abstract syntax includes semantic elements as well. The integrity constraints, which define well-formedness rules for the models, are frequently identified as "static semantics". The *semantic domain* (*S)* is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The mapping M_C: $A \rightarrow C$ assigns concrete syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The semantic mapping M_S: $A \rightarrow S$ relates syntactic constructs to those of the semantic domain. The definition of the (DSM) language proceeds by constructing metamodels of the language (to cover *A* and *C*), and by constructing a metamodel for the semantics (to cover M_C and M_S).

One key aspect of the model-driven development (and in particular, MIC) is that *model-driven concepts should be recursively applied* [L]. This means that one should use models (and modeling languages) to define the DSMLs, and the transformations on those languages, and thus models should be used not only in the (domain-specific) work, but also in the engineering of the development tool suite itself. In other words, models should drive the construction of the tools. This recursive application of the model-driven paradigm leads to a unifying approach, where the tools are built using the same principles and techniques as the (domain) applications. Furthermore, one can create generic, domain-independent tools that could be customized (via models) to become domain-specific tools, in support of domain-specific models.

Models that define DSML-s are *metamodels*, and thus a metamodeling approach should support the definition of the concrete and abstract syntax, as well as the two mappings mentioned above. Obviously, the metamodels have a language, with an abstract and concrete syntax, etc. and this language is *recursively* defined, using itself. Thus, the metamodeling language is defined by a metamodel, in the metamodeling language – thus closing the recursion.

Through experience we have learned that *the primary issue one has to address in defining a DSML is that of the abstract syntax* [L]. It is not surprising, as abstract syntax is closely related to database schemas, conceptual maps, and alike that specify the core concepts, relationships and attributes of systems. Note that the abstract syntax imposes the inherent organizational principles of the domain and all other ingredients of a DSML are related to it.

We have chosen the concrete syntax of UML class diagrams to define the abstract syntax, as it is widely known, well-documented, and sufficiently precise. *When choosing a concrete syntax for a DSML it is important to use one that is familiar to the domain engineers* [L], in this particular case the language developers. A UML class diagram defines a conceptual organization for the domain, but also the data structures that can hold the domain models. This mapping from the class diagrams to data (class) structures has been implemented in many systems.

As discussed above, the definition of abstract syntax must include the specification of well-formedness rules for the models. We have chosen the well-documented OCL approach here: OCL constraints could be attached to the metamodel elements and they constrain the domain models. Note the difference: in conventional UML constraints restrict the object instances; here *the meta-level DSML constraints restrict the models (which are, in effect, instances of the classes of the abstract syntax)* [L].

One important issue with constraints is when and how they are used. As our constraints refer to the domain models, they are evaluated when those domain models are constructed and manipulated. At the time when domain models are constructed the modeler can invoke a 'constraint checker' that verifies whether the domain models comply with the well-formedness specified in the metamodel. Occasionally these checks are automatically triggered by specific editing operations, but most often the modeler has to invoke them. Often the checks are made just before a model transformation is executed on the models. *As these checks may involve complex computations, it is an interesting research question when exactly to activate them* [L]; after every editing operation, upon a specific modeler command, when the models are transformed, etc.

The *concrete syntax* defines the rendering of the domain model in some textual or graphical form. Obviously, the *abstract syntax can be rendered in many different concrete forms* [L], and different forms could be effective for different purposes. A purely diagrammatic form is effective for human observation, but an XML form is better for automated processing. *Choosing a concrete syntax has a major impact on the usability of a DSML* [L].

There have been a number of successful research efforts to make the concrete syntax highly flexible [17][37]. These techniques typically provide a (frequently declarative) specification for rendering the abstract syntax into concrete syntax as well as interpreting elementary editing events as specific operations that manipulate the underlying data structures of models. Alternatively, one can generate diagram editors from specifications [35]. These techniques are very flexible and general and can be used to create very sophisticated model editing environments. In effect, these techniques *operationalize* the mapping M_C above.

We have chosen a different approach that is less flexible but allows rapid experimentation with DSML-s; we call this 'idiomatic specification of concrete syntax' [25]. In our previous work, we have created a number of graphical modeling environments (graphical model editors) that have used a few model organization principles coupled together with a few visual rendering and direct manipulation techniques. For example, hierarchical containment, simple associations between model elements, associations between elements of disparate models that are contained within higher order models, and indirect referencing are such model organization principles that could be visualized using hierarchical diagrams, edges between icons, edges between ports of icons, and icons that act as pointers to distant model elements, respectively. Each such model organization principle is represented with a *visual idiom*. In our metamodeling language, each metamodel element has a stereotype that indicates the visual idiom to be used when rendering the corresponding domain model element. This approach, while much more limited than the approaches to relating concrete syntax to abstract syntax mentioned above, gives a rapid feedback for the designer of a DSML: the designer constructs UML class diagrams using predefined classes and associations with predefined stereotypes (e.g. <<Atom>>, <<Model>>, <<Connection>>, <<Reference>>, <<Set>>, etc.) and the resulting diagram immediately specifies not only the abstract syntax of the DSML, but also the concrete syntax. With the help of a generic visual modeling environment, one can experiment with the new DSML literally within seconds. This experience has shown that *choosing a simple*

technique for specifying the visualization of models could be very effective, although much less general than a full realization of the mapping M_C: $A \rightarrow C$ [L].

Defining the *semantics* of a domain-specific modeling language is a complex problem, and it is the subject of active research. In our MIC suite we have chosen a transformation based approach that we will discuss in a subsequent section.

Related work: Model-driven development is outlined in the Model-Driven Architecture vision of OMG, and it is an active area of research as illustrated by the series of conferences on 'Model-Driven Engineering Languages and Systems'. However, software engineering environments for model-driven systems development have evolved from classical integrated development environments [32], and many of the same problems appear (and are re-solved) in a newer setting. Arguably, the novelty in MDA/MDE is the use of codified modeling languages for object-oriented design (UML) and the increasing use of domain-specific abstractions and dedicated, visual, domain specific modeling languages [26]. As such, the focus in the model-driven approach is moving away from the classical (textual) 'document' oriented approach towards to (graphical) 'model' oriented approach. This has an implication on how the development artifacts are stored and manipulated: in classical text-oriented environments parsing and un-parsing are important steps, while in model-driven environments (graphical) models are often rendered graphically and manipulated directly. Interestingly, one can draw parallels between the data model for abstract syntax trees (for source code) and the metamodels: they capture the concepts, relationships, and attributes of a 'language' (for programming and modeling, respectively).

3 Model Transformations

Model transformations play an essential role in any model-driven development toolchain, as discussed above [35]. They integrate different tools, they are used in refactoring and evolving model-based designs, they were used to specify code generators, and they are used in everyday work, for rapid development activities. Additionally, their efficiency, quality, and robustness are of great importance for pragmatic reasons: inefficient transformations lengthen development iterations, poor quality transformations produce inefficient models or code, and brittle transformations can cause great frustrations among developers.

It is widely recognized in the model transformation community that graph transformations serve as a suitable foundation for building model transformation systems. Graph transformations are not the only approach, but because of their long history and solid mathematical foundations they provide a solid background upon which model transformation systems can be built.

3.1 Model Transformations via Efficient Graph Transformations

Graph transformations are specified in the form of graph rewriting rules, where each rule contains a left-hand-side graph (LHS) and right-hand-side graph (RHS) [39]. When a rule is applied, an isomorphic occurrence of the LHS in the input graph is

sought and when found it is replaced by the RHS of the rule. Typically a transformation consists of more than one rule and these are applied in some order, according to some specification (either implicit or explicit). Note that the input and the output of the transformation are typed graphs, where each node has a specific type, and a rule matches only if node types match between the LHS (the pattern) and the input (host) graph.

Graph rewriting rules offer a very high-level formalism for defining transformation steps. It is easy to see that the procedural code that performs the same function as a rewriting rule could be quite complex. In fact, every graph rewriting rule execution involves a subgraph isomorphism search, followed by the manipulation of the target (output) graph. *The efficiency of the graph transformation is thus highly dependent on the efficiency of the graph isomorphism search* [L].

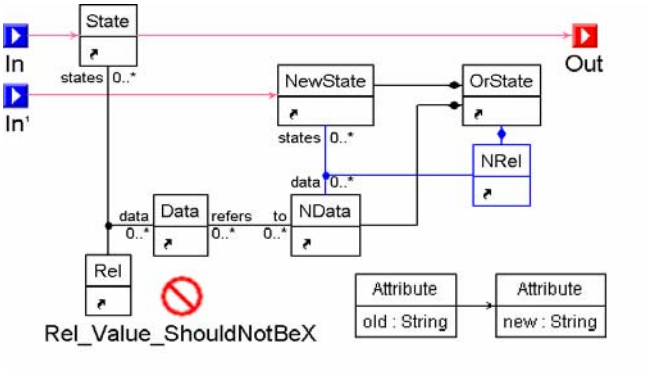


Fig. 1. Model transformation rule example

The worst-case run-time complexity of graph isomorphism test is exponential in the size of the graphs involved, but in graph transformations we only search for a fixed pattern, and the worst case time complexity for is $O(n^k)$, where n is the size of the graph and k is the size of the pattern. In our graph transformation-based model transformation system, called GReAT [2], we further reduced this by using localized search. The host graph is typically much larger than the pattern graph, and the pattern matches in a local neighborhood of a relative small number of nodes. Such localized search can be achieved by pre-binding some of the nodes in the pattern to specific nodes in the host graph. As shown on Fig. 1, the `State` and `NewState` nodes of the pattern are bound to two nodes in the graph (`In` and `In`^{1}, respectively), before the rest of the pattern is matched. In other words, the pattern is not matched against all nodes in the host graph, rather only in the neighborhood of selected, specific nodes. When the rule is evaluated, the pattern matcher produces a set of bindings for the pattern nodes `Rel`, `Data`, `NData`, and `OrState`, given the fixed binding of the nodes `State` and `NewState`. Starting the search from 'pivot' nodes leads to significant reduction in the complexity of the pattern matching process as the size of the local context is typically small (provided one avoids the so-called V-structures [11]).

Localized search, however, necessitates the determination of the locale, i.e. the binding of the `State` and `NewState` nodes in the example. This problem was solved by recognizing the need for a traversal path in the input graph. General purpose graph transformation approaches perform graph matching on the entire graph, and this has serious implications on the execution speed of such systems. In our system, similarly to some other systems like PROGRES [41] and Fujaba [25], we require the developer to explicitly provide a traversal path that sets how the rewriting rules are applied in the input graph. In practical systems model graphs have a well-known 'root' node where traversal can start form, and the first step in the transformation must have a rule that binds that root to one of the pattern nodes. A rewriting rule can also 'hand over' a node (existing, i.e. matched or newly created) to a subsequent rule (this is indicated on the example by the connection from the `State` to the `Out` port of the rule). Note that the patterns expressed in the rewriting rules and the sequencing of the rules (i.e. connecting the output ports of rules to input ports of other rules) implicitly specify how the input graph is traversed (and thus how the rewriting operations are sequenced). The sequencing can be combined with a hierarchy, as shown on Fig. 2.

While the approach appears to be more complex (and 'lower-level' compared to general graph transforms), in practice it is quite manageable. Depth-first and breadth-first traversals, traversals using arbitrary edge types, even fixpoint iterations over the graph are straightforward to implement. In our experience, *trading off generality and developer's effort for efficiency in the resulting transformation results in transformations that are not only reasonably fast (on large graphs) but also easier to understand, debug, and verify* [L].

In our research, we wanted to build 'industrial strength' model transformations that operate on large models. Our first implementation of the model transformation engine was completely 'interpreted': the engine executed the rewriting rule sequence on the input graph; exhibiting expected performance shortcomings. Once the semantics of the transformation rules and programs was clear and stabilized, we have developed a code generator that produced executable code from the transformation models. The generator was implemented using the well-known partial evaluator technique and it produced code based on the partial evaluation of the transformation program with respect to the transformation interpreter semantics. *For practical applications, such a 'compilation-based' approach to enhancing the performance of model transformations is essential* [L].

Related work: PROGRES [41] is arguably the first widely used tool for specifying transformations on structures represented as graphs. PROGRES has sophisticated control structures for controlling the rewriting the process, in GReAT we have used a similar, yet different approach: explicitly sequenced rules that form control flow diagrams. PROGRES also supports representing type systems for graphs; in GReAT we use UML diagrams for this purpose. The very high-level notion of graph transformations used in PROGRES necessitates sophisticated techniques for efficient graph matching ([10] [39]); in GReAT we mitigate this problem by using less powerful rules and (effectively) performing a local search in the host graph.

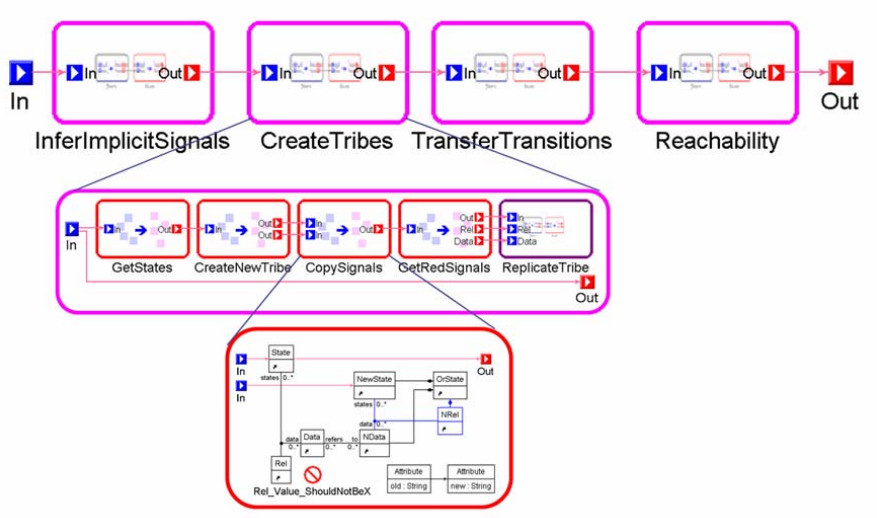


Fig. 2. Sequencing and hierarchy of rewriting rules

Fujaba [25] is similar to GREAT in the sense that it relies on UML (the tool was primarily built for transforming UML models) and uses a technique for explicitly sequencing transformation operations. Fujaba follows the state-machine-like "story diagram" approach [13] for scheduling the rewriting operations; a difference from GReAT.

AGG [43] is a graph transformation tool that relies on the use of type graphs, similar to UML diagrams but does not support all UML features (e.g. association classes). Recent work related to AGG introduced a method for handling inheritance, as well as a sophisticated technique for checking for confluence (critical pair analysis). In GReAT, inheritance is handled in the pattern matching process, and the confluence problem is avoided by using explicit rule sequencing. AGG has support for controlling the parsing process of a given graph in the form of layered grammars; a problem solved differently in GReAT.

VIATRA [5] is yet another graph transformation tool with interesting capabilities for controlling the transformations (state machines), and the composition of more complex transformations. In GReAT similar problems were addressed via the explicit control flow across rules and the formulation of blocks. Higher-order transformations were also introduced in VIATRA; there is no similar capability in GReAT currently.

GReAT can also be compared to the recent QVT specification [39] of the OMG MDA standardization process. However, we should emphasize that GReAT was meant to be a research tool and not an industry standard. With respect to the QVT, the biggest difference between GReAT and QVT is in the highly declarative nature of the QVT: it focuses on relation mappings. This is a very high-level approach, and it is far from the pragmatic, lower-level, efficiency-oriented approach followed in GReAT. We conjecture that describing a transformation in QVT is probably more compact, but the same transformation in GReAT is more efficient.

In a more general setting, we should compare GReAT and the tool environment it belongs to, i.e. the MIC tools including GME and UDM. Honeywell's DOME [13], MetaCASE's MetaEdit [25], and the ATOM3 [28] environment are the most relevant examples that support domain-driven development. The main difference between them and the MIC tools is in the use of UML and OCL for metamodeling and the way the metamodels are (also) used for instantiating a visual modeling environment. Also, our transformations follow a high-level method for describing the transformation steps expressed in the context of the metamodels. With exception of ATOM3, all the above tools use a scripting language, in contrast.

3.2 Practical Use of Model Transformations

The model transformation environment we have created has been used in a number of academic and practical projects. Students, researchers, and developers have used it to create practically useful transformations ranging from converting between modeling formalisms to generating code from models. Some of the transformations were of the 'once-only' (or 'throw-away') type; some of them are in daily use in tools. In these efforts we have learned a few important lessons discussed below.

1. **Reusable patterns.** *Given a model transformation language, developers often discover important and useful transformation patterns that are worth documenting and reusing* [L]. These patterns are essentially generic transformation algorithms that are usable across a number of domain specific modeling languages. Conceptually, they are like C++ template libraries that provide generic data structures and algorithms over domain-specific types. Practically, they provide a reusable solution to a recurring transformation problem. Such patterns are rarely implemented by a single rule, rather, by a sequence or group of rules.

   To increase the reusability of such transformation patterns, a model transformation language should support templates [L], which are rules or rule sequences that are parameterized with types. When the transformation designer wishes to use a transformation template, s/he can bind the type parameters to concrete, domain-specific model element types and the tool environment should instantiate the pattern.
2. **Cross-domain links.** In model transformations the source and target models typically (but not always) belong to different metamodels (i.e. different type systems). During the transformation process it is often necessary to create a link between two model elements that belong to different domains (metamodels), but this brings up the question: which metamodel does the association belong to? Neither of the source or target metamodels 'owns' such an association, the association belongs to the cross-product space of the two. Hence, *the model transformation system should be able to allow such 'cross-domain' links* [L]; at least temporarily, while the transformation is being executed [2].
3. **Global context.** The localized rewriting approach described above has a practical shortcoming: the context of the rewriting has to be always present during the execution of a rule [41]. That is, a rule cannot just create an 'orphan' target model element – the element has to be inserted into an appropriate container, which is in the target context. In other words, the state of the transformation

often has to be incrementally built and passed from rule to rule. This leads to rules that have superfluous input and output ports, just for passing the context through; and a large number of connections between rules. *The simplicity of the transformation model is important, and such 'accidental complexity' confuses developers* [L]. We have introduced the concept of a 'global container,' where new temporary model elements can be created and latter found via search. In other words, a rule can create a model element in this container and a subsequent rule can refer to it simply referring to the container and using the pattern matcher to find the element. *Note that such global containers are useful, although somewhat 'unhygienic' tools for implementing model transformations* [L].

4. **Multiple matches and sorting.** A model transformation rule often matches to multiple isomorphic subgraphs in an input graph, and even a localized rewriting rule could generate multiple, consistent matches with different bindings to pattern nodes. In general, the order of such matches is nondeterministic as it depends on how the underlying 'model database' is implemented. We chose to process these matches sequentially, and for every match we execute the right hand side of the rule, which leads to non-deterministic results. We found that in many applications *the order of processing of such matches does matter* [L]. To solve this problem we have introduced an optional 'sort' function for the rewriting rule that the designer can specify [41]. This function is applied to the result of the pattern matcher before the rule is actually executed, and the function can sort the results in any order of interest. *How the matches need to be sorted is domain-specific, hence it is better left in the hands of the developer* [L].
5. **Multiple matches and grouping.** When the pattern matcher generates a collection of matches (each one with a distinct set of bindings of input graph nodes to pattern nodes), the rewriting rule processes them one by one. *The major limitation with this simple algorithm is the inability to apply a single rule action across multiple matches* [L]. After all matches are computed, the rule's action (RHS) is executed individually, on each match; furthermore, there exists no mechanism by which one can access information about an earlier match while processing a specific match. This can sometimes pose a severe limitation to the types of transformations one can write. For instance, the user may need the ability to operate on an entire subgraph (composed from multiple matches) as a whole rather than on individual elements. If this subgraph may contain an arbitrary number of elements, then the graph pattern cannot be specified as a simple rule.

   We have introduced a higher-order 'subgroup' operator that allows forming groups from the matches during rule execution [3] [4]. The operator has a number of attributes the designer can specify, including functions that are evaluated to determine whether a match belongs to a group or not. The operator extends the rule execution semantics as follows: (1) the pattern matcher produces a set of matches; (2) matches are used to form groups, based on functions supplied with the operator, (3) the rule is executed for each group formed. Note that a group may contain one or more matches. *The subgroup operator has demonstrated the value of higher-order operators in rewriting rules that can operate across multiple individual rewriting steps* [L].

The above extensions have come up in practical transformation problems, and they showed that while graph transformations provide a powerful theory, when applied to model transformations they need to be specialized and adapted to pragmatic goals.

3.3 The Issue of Verification

The quality of a model-driven software development toolchain is determined by the quality of the tools it includes and the model transformations that connect together these tools. Considering that elements in the toolchain could support verification (on the code or on the model level), the *verification of the transformations, i.e. graph transformations, is of great importance* [L]. Simply, the correctness of the transformation is necessary in order to decide that the result of an automated verification applies to the original input (model), and to the code generated from the model.

Our goal was to decide the correctness of a model transformation through some verification process. The verification of model transformations is closely related to the verification of compilers – one of the great challenges of computer science. Arguably, the verification of model transformations is simpler, as the domain-specific modeling languages are often simpler and have a simpler semantics than general purpose programming languages.

One important observation is that the notion of correctness is not absolute, but it has to be defined with respect to some specific domain property, which is of interest to the users of the toolchain. For example, a model transformation can be called correct with respect to the behaviors generated by the source and the target models. For instance, the transformation is correct if the source model (with its own source semantic domain) generates the same behaviors as the target model (with its own target semantic domain). Alternatively, a transformation is correct when a property of the source model holds if and only if another property holds for the target model.

Practical model transformations are often very complex and the formal proof of correctness requires a major effort. Note that a formal proof shows that the given model transformation is correct (w.r.t. some property), for *any* input. Another feasible approach is that the proof is constructed for a particular *run* (or 'instance') of the transformation, i.e. for a given input. This, *instance-based verification of the transformation appears to be much simpler and feasible* [L]. While the concept has been developed in the context of program generators [5], we have successfully applied it to model transformations [13].

Constructing the verification for a transformation instance requires building a tool that checks what the transformation did and verifies it independently. These checks must be simple and easily verifiable. Note that this concept is similar to provers and proof checkers: the proof checking is typically much simpler than constructing the proof. For a model transformation one needs (*1) to choose the property the correctness is defined for, (2) to discover how this property can be verified from data collected during the run of the transformation, (3) to modify the model transformation to generate the data during the run, and (4) to develop (and verify) the algorithm that checks the data and thus verifies the property* [L].

One example for such transformation and verification property includes a transformation between two transition system formalisms and a state reachability property. In this case the transformation needs to generate a map that links source and target states, and the checking algorithm must verify that there is a correspondence: a strong bisimilarity between the two transition systems, hence reachability properties verified for one do hold for the other [13].

Related work: The MOF 2.0 Query / View / Transformation specification [39] provides a language for declaratively specifying transformations as a set of relations that must hold between models. A relation is defined by two or more domains, and is declared either as *Checkonly*, meaning that the relation is only checked, or *Enforced*, meaning that the model is modified if necessary to satisfy the relation. It is augmented by a *when* clause that specifies under what conditions the relation must hold, and a where clause that specifies a condition that must be satisfied by all the model elements participating in the relation. Our approach provides a solution similar to the *Checkonly* mode of QVT relations. The main difference is our use of pivot nodes to define correspondence conditions and the use of cross links. This allows us to use a look up table to match corresponding nodes. Our approach takes advantage of the transformation framework to provide a pragmatic and usable verification technique that can ensure that there are no critical errors in model instances produced by automated transformations. Triple Graph Grammars [41] can be used to represent the evolution of a model graph by applying graph transformation rules. The evolving graph must comply with a graph schema at all times. This graph schema consists of three parts, one describing the source metamodel, one describing the target metamodel, and one describing a correspondence metamodel which keeps track of correspondences between the other two metamodels. Triple graph grammar rules are declarative, and operational graph grammar rules must be derived from them. The correspondence metamodel can be used to perform a function similar to the cross links used here. This provides a framework in which a map of corresponding nodes in the instance models can be maintained, and on which the correspondence conditions can be checked. This makes it suitable for our verification approach to be applied. Some ideas on validating model transformations are presented in [28] and [29]. In [28], the authors present a concept of rule-based model transformations with control conditions, and provide a set of criteria to ensure termination and confluence. In [29], the authors focus on the syntactic correctness of rule-based model transformations. This validates whether the source and target parts of the transformation rule are syntactically correct with respect to the abstract syntax of the source and target languages. These approaches are concerned with the functional behavior and syntactic correctness of the model transformation. We focus on the semantic correctness of model transformations, addressing errors introduced due to loss or misrepresentation of information during a transformation. It is possible for a transformation to execute completely and produce an output model that satisfies all syntactic rules, but which may still not have accomplished the desired result of porting essential information from the

source model to the output model. Our approach is directed at preventing such semantic errors. Ehrig et. al. [13] study bidirectional transformations as an approach for preserving information across model transformations. They use triple graph grammars to define bidirectional model transformations, which can be inverted without specifying a new transformation. Our approach offers a more relaxed framework, which will allow some loss of information (such as by abstraction), and concentrates on the crucial properties of interest. We also feel that our approach is better suited for transformations involving multiple models and attribute manipulations.

3.4 Transformations in Evolution and Adaptation

One of the crucial properties of software systems is the need for their evolution and adaptation. Software must evolve, as new requirements arise, as flaws need to be fixed, and as the system must grow in its capabilities. *Model-driven development toolchains are also software systems, and hence the same requirement applies: they need to evolve and adapt* [L]. The problem is especially acute for tools that use domain-specific modeling languages, as the DSML-s may evolve not only from project to project, but often during the lifetime of one project.

The issue of evolution for a DSML is not only how the language changes, but also what effect this has on the already existing models. Specifically, if a large number of models have already been built with a DSML of version *n*, how do we use these models with DSML version *n+1*, etc.? The problem is related to schema evolution (i.e. how we evolve database content when the schema evolves), but modeling languages typically have much richer semantics and consistency constraints than typical database schemata. *For DSML-s the language evolution problem is essentially a model migration problem, i.e. how to migrate models when the DSML evolves* [L].

The problem can be cast as a model transformation problem, i.e. how can one create model transformations that automatically migrate the models from one DSML version to the next. To analyze this problem we need to recall how a DSML is defined; i.e. the metamodels. In this paradigm, the DSML evolves via changes applied to the metamodels; i.e. changes in the abstract and concrete syntax, in the well-formedness constraints, and in the semantic mapping. As model transformations are anchored in the abstract syntax of the DSML it is natural to consider the metamodel changes on that level. Changes in the concrete syntax do not affect the models (until a syntax-free realization of the models exists), while changes in the well-formedness constraints and semantic mapping could possibly be also formulated as a model transformation problem. These latter two cases could be formulated as posing the question: how shall the models be transformed that they comply with the updated well-formedness constraints (if at all) and how they shall be transformed such that they preserve their semantics under the updated semantic mapping?

Changes in the abstract syntax part of the metamodel involve changing the UML class diagrams representing that. Such changes can be captured as elementary editing operations on the diagram, including adding, removing, and modifying classes and associations, etc. But focusing on these low-level changes makes it exceedingly hard to discover the (meta-) modeler's intent. For instance removing a class called `Failure` and adding a class `Fault` may miss the point that this is a simple renaming of a class

without changing the semantics. Hence, *evolution in the abstract syntax cannot be dependably deduced by observing editing changes on the metamodels; the modeler needs to provide guidance or explanations for such changes* [L]. For pragmatic reasons, naturally, only the changes need to be documented (or discovered by some automation, if feasible) – *parts of the metamodels that did not change should not become subject to model transformations* [L].

Such analysis lead us to a simple (graphical) language that allows the modeler to document the metamodel changes by capturing how 'old' metamodel elements are related to 'new' metamodel elements [32]. Note that the modeler essentially supplies rewriting rules that proscribe how a model migration engine should convert 'old' models into 'new' models. In this graphical language, called Model Change Language (MCL), we have defined a number of idioms for representing prototypical cases for metamodel changes (and thus model migration). Fig. 3 illustrates the major idioms of the language – these have been discovered through practical experience with model migration problems. While the use of these idioms has been proven useful in specific model migration problems, the formal semantics of MCL is subject of active research. Naturally, model evolution is not a solved problem yet, but transformations appear to offer interesting opportunities.

A migration rule specified in MCL describes a migration step which is centered on a single, specific 'old' metamodel element that dominates the rule. The semantics of the migration rule is as follows: whenever the dominant model element is found in the 'old' model and the left-hand side of the rule matches; then execute the migration as specified (e.g. create a 'new' model element, etc.). If an 'old' model element is encountered that is not mapped by a migration rule then check if there is a 'new' metamodel element with the same name and create that in the 'new' model, if there is none then give a warning that an 'old' model element was encountered but could not be mapped. Note that there are explicit rules for saying that some 'old' model elements need to be removed because there are no corresponding 'new' model elements – this allows detecting that the migration of some model elements was not specified correctly by migration rules.

In MCL we faced the problem of limiting the scope of the search and we found a solution that appears to work well. The solution is based on the observation that *model databases mostly follow a tree structure, and a dominant spanning tree can be found for the model graph* [L], often via the model containment hierarchy. Hence, we first use a depth-first traversal on the model tree, visiting every node in the graph and trying to find a migration rule. The rule semantics briefly described above is applied, when possible. However, there could be rules that are not applicable yet, because they depend on model elements that have not been visited and processed yet. These rules are pushed onto a queue of delayed rules and the traversal continues. Once the depth first traversal terminates, we keep processing the delay queue until it becomes empty. This simple, fixed traversal strategy works surprisingly well. Arguably, *for practical model-driven systems that use hierarchical organization model transformations can efficiently be performed using a depth-first traversal, followed by the processing of rewriting steps that had to be delayed during the first traversal* [L].

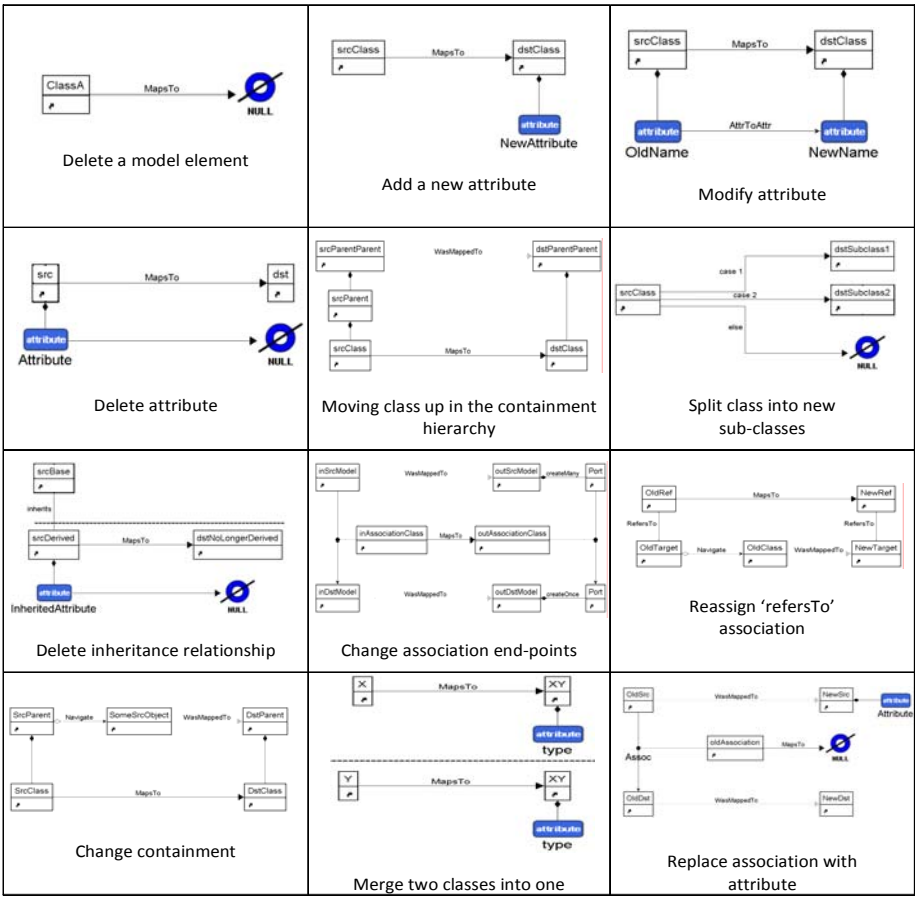


Fig. 3. Some idioms of the Model Change Language

Related work: Our work on model migration has its origins in techniques for database schema evolution [5]. More recently, though, even traditional programming language evolution has been shown to share many features of model migration. Drawing from experience in very large scale software evolution, [15] uses several examples to establish analogies between tradition programming language evolution and metamodel and model co-evolution. Using two industrial metamodels to analyze the types of common changes that occur during metamodel evolution, [17] gives a list of four major requirements that a model migration tool must fulfill in order to be considered effective: (1) Reuse of migration knowledge, (2) Expressive, custom migrations, (3) Modularity, and (4) Maintaining migration history. The first, reusing migration knowledge is accomplished by the main MCL algorithm: metamodel independent changes are automatically deduced and migration code is automatically generated. Expressive, custom migrations are accomplished in MCL by (1) using the metamodels directly to describe

the changes, and (2) allowing the user to write domain-specific code with a well-defined API. Our MCL tool also meets the last two requirements of [17]: MCL is modular in the sense that the specification of one migration rule does not affect other migration rules, and the history of the metamodel changes is persistent and available to facilitate model migration. [8] performs model migration by first examining a difference model that records the evolution of the metamodel, and then producing ATL code that performs the model migration. Their tool uses the difference model to derive two model transformations in ATL: one for automatically resolvable changes, and one for unresolvable changes. They note that the generated transformation that deals with the unresolvable changes must be refined by the user, but details of how to accomplish this refinement are not provided. Also, [7] does not specify exactly how the difference models are calculated, only that they can be obtained by using a tool such as `EMFCompare`. MCL, on the other hand, uses a difference model explicitly defined by the user, and uses its core algorithm to automatically deduce and resolve the breaking resolvable changes. Changes classified as breaking and unresolvable are also specified directly in the difference model, which makes dealing with unresolvable changes straightforward: the user defines a migration rule using a graphical notation that incorporates the two versions of the metamodel and uses a domain-specific C++ API for tasks such as querying and setting attribute values. In [7], the user has to refine ATL transformation rules directly in order to deal with unresolvable changes. [17] describes the benefits of using a comparison algorithm for automatically detecting the changes between two versions of a metamodel, but says they cannot use this approach because they use `ECore`-based metamodels, which do not support unique identifiers, a feature needed by their approach. Rather than have the changes between metamodel versions defined explicitly by the user, they slightly modify the `ChangeRecorder` facility in the EMF tool set and use this to capture the changes as the user edits the metamodel. Their migration tool then generates a model migration in the Epsilon Transformation Language (ETL). In the case that there are metamodel changes other than renaming, user written code in ETL to facilitate these changes cannot currently be linked with the ETL code generated by their migration tool. In contrast to this, MCL allows the user to define complex migration rules with a straightforward graphical syntax, and then generates migration code to handle these rules and links it with the code produced by the main MCL algorithm. [10] presents a language called COPE that allows a model migration to be decomposed into modular pieces. They note that because metamodel changes are often small, using endogenous model transformation techniques (i.e., the metamodels of the input and output models of the transformation are exactly the same) can be beneficial, even though the two metamodels are not identical in the general model migration problem. This use of endogenous techniques to provide a default migration rule for elements that do not change between metamodel versions is exactly what is done in the core MCL algorithm. However, in [19], the metamodel changes must be specified programmatically, as opposed to MCL, in which the metamodel changes are defined using a straightforward graphical syntax. Rather than manually changing metamodels, the work in [45] proposes the use of QVT relations for evolving metamodels and raises the issue of combining this with a method for co-adapting models. While this is an interesting idea, our MCL language uses an explicit change language to describe metamodel changes rather than model transformations.

Evolution of a DSML (and the subsequent migration of domain models) is not the only activity the toolchain users face. They often need to evolve, adapt their designs by changing models. In object-oriented software development perhaps the most powerful concept for design adaptation is the use of design patterns. By definition, design pattern is a general reusable solution to a commonly occurring problem in software design. When developers use a design pattern they modify their designs according to the pattern, in other words they instantiate the design pattern in the context of their work. If the design is captured in a model, then a design pattern is a particular arrangement of newly built or existing model elements, possibly with some new features added. Arguably, *a design pattern can be modeled as a specialized model transformation rule that rewrites a design into a new design with the design pattern features (model elements, attributes, etc.) added* [L]. Note also that design patterns applied in domain-specific modeling languages will have domain-specific elements hence they can be called as 'domain-specific design patterns'.

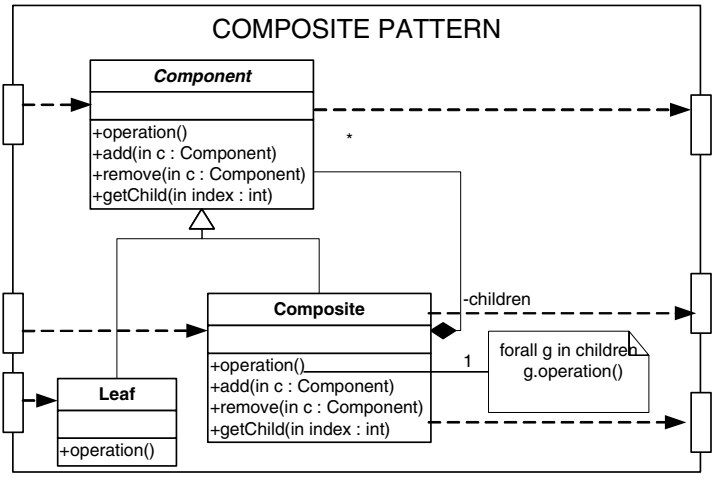


Fig. 4. The Composite Pattern as a model transformation rule

One example for realizing a design pattern as a transformation rule is shown on Fig. 4. Here the well-known 'Composite' design pattern is used. When the designer wants to introduce this pattern into a design, s/he will either just copy it into the model and modify it, or bind it to existing model elements (say, Compound and Primitive, shown on the left) which will result in a modified model that contains the original as well as new elements (shown on Fig. 5).

Note that the application of design patterns becomes an interactive activity this way that the modeler performs at model construction time. Design patterns can be applied to existing design, and they can extend or even refactor those designs. Design patterns can be highly domain specific hence they can be applied in any DSML.

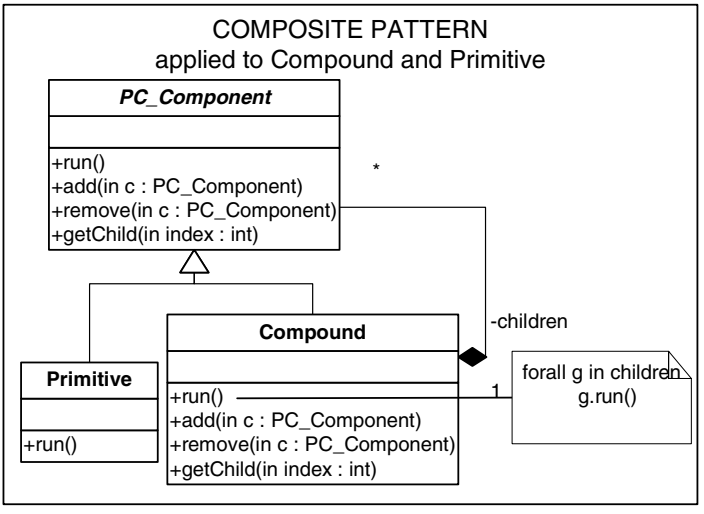


Fig. 5. Composite pattern applied

We have created a set of tools to support the definition and application of design patterns in arbitrary DSML-s that are defined by a metamodel [32]. One tool is used (once) to extend the metamodel of a given DSML such the patterns can be built from existing model elements. Another tool is available to the modeler that uses the DSML: this tool applies the pattern as a local model transformation. The modeler can bind existing model elements to elements of the pattern and the applicator tool extends and modifies the model as specified by the pattern.

Related work: There are several mainstream tools that support UML design patterns, or describe design patterns using general-purpose languages, as opposed to using the metamodel of the DSMLs. Moreover, there are several approaches for pattern formalization. Here, we reference the closest related work only. Previous work [32] has justified the demand for Domain-Specific Model Patterns by contributing several DSMLs. Moreover, it describes relaxation conditions for the metamodels in order to make metamodeling environments support the editing of incomplete models. As opposed to the approach introduce above, it deals with static model patterns only. In our approach, relaxations can be made on the metamodel of the pattern environment. The multiplicities can be substituted with the upper bound of the multiplicity set, dangling edges can be defined with ignored end nodes and transitive containment can be solved with ignored containers. Incomplete attributes can be implemented the same way. [17] describes a UML-based language, namely, the Role-Based Metamodeling Language (RBML), which is able to specify domain-specific design patterns. This approach treats domain patterns as templates, where the parameters are roles, and a tool generates models from this language. Compared to our approach, the paper [22] proposes a formal way to specify the pattern embedding for the static aspect. The behavioral formalization is closely coupled with design patterns defined in UML. The work described in [5] formalizes the embedding, tracing, and synchronization between several

pattern aspects that may be defined in different languages. These results constitute an excellent theoretical formalization of the tracing aspects for model patterns defined in the static aspect.

4 Summary and Conclusions

The model-driven development approach has significantly changed how software is built and evolved, and new development environments are coming equipped with model-driven support. The techniques and the tools we have developed in the past decade indicate that model-driven development works, but the complexity of the development tools (and the effort to build them) is increasing as well. In this paper we have outlined a few of the lessons that we have learned during building and using our tools on non-trivial projects. Building tools to build software is essential to solve the software development problem and the effort put into constructing a good tool (-suite) pays off in developer's productivity. The lessons described in this paper show steps in an evolutionary process, and by no means should be considered the final word on model-driven development. As tools and techniques evolve, we need to learn new lessons, and enable the developers to benefit from them.

Acknowledgements. This work was sponsored, in part, by the Evolutionary Design of Complex Systems and Software, the Model-based Integration of Embedded Systems, and the Software Producibility programs of DARPA and AFRL, and by the NSF ITR on "Foundations of Hybrid and Embedded Software Systems". The views and conclusions presented are those of the authors and should not be interpreted as representing official policies or endorsements of DARPA, NSF, or the US government. The work described in this paper has involved a number of researchers, engineers and students at the institution of the author, including but not limited to: Janos Sztipanovits, who started it all, Aditya Agrawal, Arpad Bakay, Daniel Balasubramanian, Jeff Gray, Zsolt Kalmar, Akos Ledeczi, Tihamer Levendovszky, Endre Magyari, Miklos Maroti, Anantha Narayanan, Benjamin Ness, Sandeep Neema, Feng Shi, Jonathan Sprinkle, Ryan Thibodeaux, Attila Vizhanyo, Peter Volgyesi. All this is really their work.

References

1. Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G.: Reusable Idioms and Patterns in Graph Transformation Languages. Electronic Notes in Theoretical Computer Science 127, 181–192 (2005)
2. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The Design of a Language for Model Transformations. Journal on Software and System Modeling 5, 261–288 (2006)
3. Balasubramanian, D., Narayanan, A., Neema, S., Shi, F., Thibodeaux, R., Karsai, G.: A Subgraph Operator for Graph Transformation Languages. ECEASST 6 (2007)
4. Balasubramanian, D., Narayanan, A., Neema, S., Ness, B., Shi, F., Thibodeaux, R., Karsai, G.: Applying a Grouping Operator in Model Transformations. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 410–425. Springer, Heidelberg (2008)

5. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In: Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data, pp. 311–322 (2007)
6. Bottoni, P., Guerra, E., Lara, J.: Formal Foundation for Pattern-Based Modelling. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 278–293. Springer, Heidelberg (2009)
7. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, ECOC, pp. 222–231 (2008)
8. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D.: VIATRA — Visual Automated Transformations for Formal Verification and Validation of UML Models. In: IEEE Conference on Automated Software Engineering, pp. 267–270 (2002)
9. Denney, E., Fischer, B.: Certifiable Program Generation. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 17–28. Springer, Heidelberg (2005)
10. Dörr, H.: Efficient Graph Rewriting and its implementation. LNCS, vol. 922. Springer, Heidelberg (1995)
11. Dörr, H.: Bypass Strong V-Structures and Find an Isomorphic Labelled Subgraph in Linear Time. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 305–318. Springer, Heidelberg (1995)
12. DSLs: The Good, the Bad, and the Ugly, Panel at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Nashville, TN, October 22 (2008)
13. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Fundamental Approaches to Software Engineering, pp. 72–86 (2007)
14. Engstrom, E., Krueger, J.: Building and rapidly evolving domain-specific tools with DOME. In: IEEE International Symposium on Computer-Aided Control System Design, pp. 83–88 (2000)
15. Favre, J.M.: Meta-models and Models Co-Evolution in the 3D Software Space. In: Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICS (2003)
16. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
17. Fondement, F., Baar, T.: Making metamodels aware of concrete syntax. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 190–204. Springer, Heidelberg (2005)
18. Gruschko, B., Kolovos, D.S., Paige, R.F.: Towards Synchronizing Models with Evolving Metamodels. In: Proceedings of the International Workshop on Model-Driven Software Evolution (MODSE) (2007)
19. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE: A Language for the Coupled Evolution of Metamodels and Models. In: MCCM Workshop at MoDELS (2009)
20. Herrmannsdoerfer, M., Benz, S., Jurgens, E.: Automatability of Coupled Evolution of Metamodels and Models in Practice. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 645–659. Springer, Heidelberg (2008)
21. Kim, D.-K., France, R., Ghosh, S., Song, E.: A UML-Based Metamodeling Language to Specify Design Patterns. In: Proc. Workshop Software Model. Eng. (WiSME) (2004)

22. Kim, S.-K., Carrington, D.A.: A formalism to describe design patterns based on role concepts. Formal Asp. Comput. 21(5), 397–420 (2009)
23. Karsai, G., Narayanan, A.: On the Correctness of Model Transformations in the Development of Embedded Systems. In: Kordon, F., Sokolsky, O. (eds.) Monterey Workshop 2006. LNCS, vol. 4888, pp. 1–18. Springer, Heidelberg (2007)
24. Karsai, G., Ledeczi, A., Neema, S., Sztipanovits, J.: The Model-Integrated Computing Toolsuite: Metaprogrammable Tools for Embedded Control System Design. In: IEEE Joint Conference CCA, ISIC and CACSD, Munich, Germany (2006)
25. Kelly, S., Tolvanen, J.-P.: Visual domain-specific modelling: Benefits and experiences of using metaCASE tools. In: Bezivin, J., Ernst, J. (eds.) Proceedings of International Workshop on Model Engineering, ECOOP 2001 (2000)
26. Kent, S.: Model Driven Engineering. In: Proceedings of the Third International Conference on Integrated Formal Methods, May 15-18, pp. 286–298 (2002)
27. Klein, T., Nickel, U., Niere, J., Zündorf, A.: From UML to Java And Back Again, Tech. Rep. tr-ri-00-216, University of Paderborn, Paderborn, Germany (September 1999)
28. Küster, J.M.: Systematic validation of model transformations. In: Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004) (October 2004)
29. Küster, J.M., Heckel, R., Engels, G.: Defining and validating transformations of uml models. In: HCC 2003: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments, Washington, DC, USA, pp. 145–152. IEEE Computer Society, Los Alamitos (2003)
30. de Lara, J., Vangheluwe, H.: AToM3: A Tool for Multi-formalism and Meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
31. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE Computer, 44–51 (November 2001)
32. Levendovszky, T., Lengyel, L., Mészáros, T.: Supporting domain-specific model patterns with metamodeling. Software and Systems Modeling (March 2009), doi:10.1007/s10270-009-0118-3
33. Levendovszky, T., Karsai, G.: An Active Pattern Infrastructure for Domain-Specific Languages. Accepted for presentation at First International Workshop on Visual Formalisms for Patterns (VFfP 2009), Corvallis, Oregon, USA (2009)
34. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996) ISBN 3-540-61985-2
35. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Sci. Comput. Program. 44(2), 157–180 (2002), http://dx.doi.org/, doi:10.1016/S0167-6423(02)00037-0
36. Model-Driven Architecture Guide, OMG, http://www.omg.org/docs/omg/03-06-01.pdf
37. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schnekenburger, R., Gérard, S., Jézéquel, J.M.: Model-driven analysis and synthesis of concrete syntax. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 98–110. Springer, Heidelberg (2006)
38. Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G.: Automatic Domain Model Migration to Manage Metamodel Evolution. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 706–711. Springer, Heidelberg (2009), http://dx.doi.org/, doi:10.1007/978-3-642-04425-0_57
39. OMG QVT specification, http://www.omg.org/docs/ptc/05-11-01.pdf

40. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific Publishing Co. Pte. Ltd., Singapore (1997)
41. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
42. Schürr, A., Winter, A., Zündorf, A.: Graph grammar engineering with PROGRES. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 219–234. Springer, Heidelberg (1995)
43. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
44. Vizhanyo, A., Neema, S., Shi, F., Balasubramanian, D., Karsai, G.: Improving the Usability of a Graph Transformation Language. In: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), March 27, 2006. Electronic Notes in Theoretical Computer Science, vol. 152, pp. 207–222 (2005)
45. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
46. Zündorf, A.: Graph pattern matching in PROGRES. In: Cuny, J., Engels, G., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1994. LNCS, vol. 1073, pp. 454–468. Springer, Heidelberg (1996)

Workflow-Driven Tool Integration Using Model Transformations *

András Balogh[3], Gábor Bergmann[1], György Csertán[3], László Gönczy[1], Ákos Horváth[1], István Majzik[1], András Pataricza[1], Balázs Polgár[1], István Ráth[1], Dániel Varró[1], and Gergely Varró[2]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
{bergmann,gonczy,ahorvath,majzik,pataric,polgar,rath,varro}@mit.bme.hu
[2] Department of Computer Science and Information Theory,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
gervarro@cs.bme.hu
[3] OptxWare Research and Development LLC,
H-1137 Katona J. u. 39.
{andras.balogh,gyorgy.csertan}@optxware.com

Abstract. The design of safety-critical systems and business-critical services necessitates to coordinate between a large variety of tools used in different phases of the development process. As certification frequently prescribes to achieve justified compliance with regulations of authorities, integrated tool chain should strictly adhere to the development process itself. In order to manage complexity, we follow a model-driven approach where the development process is captured using a precise domain-specific modeling language. Each individual step within this process is represented transparently as a service. Moreover, to carry out individual tasks, systems engineers are guided by semi-automated transformation steps and well-formedness constraint checking. Both of them are formalized by graph patterns and graph transformation rules as provided by the VIATRA2 framework. In our prototype implementation, we use the popular JBPM workflow engine as orchestration means between different design and verification tools. We also give some insights how this tool integration approach was applied in recent projects.

1 Introduction

Complex development projects, especially, in the field of safety-critical systems, necessitate the use of a multitude of software tools throughout the entire life-cycle of the system under design for requirements eliciation, design, implementation, verification and validation as well as change management activities.

However, in order to ensure safety, the verification of tool output is mandated by industrial certification standards (like DO-178B [1] for avionics systems), which requires

* This work was partially supported by the European Union as part of the MOGENTES (STREP-216679), the DIANA (AERO1-030985) and the SENSORIA (IST-3-016004) projects.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 224–248, 2010.

enormous efforts. Software tool qualification aims at reducing or even eliminating such efforts to obtain certification credit for the tool itself by ensuring that a tool always produces deterministic and correct output.

Standards differentiate between *verification tools* that cannot introduce errors but may fail to detect them, and *development tools* whose output is part of the critical system and thus can introduce errors. According to the standard, a development tool needs to be qualified to at least the same level of scrutiny as the application it is used to develop. The main functionality of a software development tool is thus to correctly and deterministically transform an input artifact into output.

Unfortunately, the qualification of software tools is extremely costly, even minor changes would require re-certification efforts [2] resulting in several man-years of work. Furthermore, qualified tools are almost exclusively relying upon closed, internal technologies of a company [3], without using external components, as vendors of safety-critical tools are unable to control the development of external components with the level of preciseness required by certification standards. Finally, the tool integration costs for building a uniform tool chain can frequently exceed the total costs of the individual tools themselves.

The extreme costs of certification and tool qualification are largely due to the fact that integration between different tools is carried out in an ad hoc way in the industry [4]. It is still a very common scenario that the output of one tool is ported manually to serve as the input of another tool. Moreover, the chaining of different tools is completely decoupled from the rigorous development processes necessitated by the standards.

In the current paper, we propose to use model transformation services organized into complex *model transformation chains*. These transformation chains are closely aligned with the designated development process as *driven by precise workflow models* where workflow activities comprise of individual development steps carried out by some tool, which creates some output artifacts (models and code, configuration files) from some input artifacts. Moreover, each step in a tool integration chain can be hierarchically refined later on by using workflows for capturing the main steps of individual development tools.

Formalizing design processes with workflow allows formal adherence checks with certification guidelines. Moreover, workflow-dirven tool integration aligns development and V&V toolchains with the actual development process itself.

Individual development steps of the transformation chain are treated as black-box components, where functionalities carried out by the tool are precisely captured by *contracts formalized as graph patterns*. This black-box approach enables that both automated and user-driven development steps can be integrated in a uniform way to the tool chain. Furthermore, *automated* tool integration or development *steps can be captured by model transformations* formalized by means of graph transformation rules.

This approach has been successfully applied to developing individual tools (as in the DECOS [5] and DIANA [6] EU projects) as well as for complete tool integration chains (as in SENSORIA [7], MOGENTES [8] and GENESYS [9] projects) in the context of safety-critical systems. Further industrialization of the framework is being carried out as part of the ARTEMIS project INDEXYS [10].

The rest of the paper is structured as follows. Section 2 summarizes the main steps and challenges of tool integration scenarios. In Sec. 3, we propose a workflow-driven approach for driving tool integration scenarios. Section 4 introduces a tool integration challenge taken from an avionics context. Section 5 discusses how graph patterns and graph transformation rules can be used in automating the development of the tool integration process. Then, in Sec. 6, we also demonstrate how such a tool integration framework has been implemented using state-of-the-art open source technologies. Finally, Section 7 discusses related work and Section 8 concludes our paper.

2 The Architecture of Tool Integration Chains

2.1 Classification of Development Activities

Complex development processes make use of a multitude of development, design, verification and documentation tools. The wide spectrum of underlying technologies, data representation formats and communication means has called for tool integration frameworks to address the need for a common underlying tool integration platform. Such middleware is typically designed to allow for various tools to be integrated as *services*, so that the integration process can be designed by concentrating on the *tasks* that are to be performed, rather than the underlying technological peculiarities.

On the conceptual level, the main functionality of each step (task) is to transform an input artifact into one or more output artifacts. This transformation view on development and tool integration tasks does not have a direct impact on the level of automation. For example, certain tasks can be either (fully) *automated*, such as compiling source code from an executable model like statecharts or running a model analysis task to reveal conceptual flaws in the design. Other development tasks are inherently *user guided* (or user driven) where a development step is completed in close interaction with the systems engineers. User guided steps typically include those where design decisions need to be made and recorded, such as modeling. While full automation is impossible (or impractical) for user guided steps, the step itself can still be interpreted using this transformational view. Moreover, automation may still implant design intelligence into such tools be performing on-the-fly validation of certain design constraints, which can reduce costs.

Development steps can also be categorized on the basis of comparing the information between the source and the target formalisms of the step.

- *Synthesis steps* (carried out by using textual or graphical editors, and even certain automated tools like schedulers, optimizers) add new information to the system under design during the completion of the step.
- *Analysis steps* (also known as verification and validation steps), on the contrary, typically abstract from existing information in order to enable checking for certain correctness properties to reveal errors in the design.
- *Derivation steps* (like code generation or model export and import with format conversion) do not add or remove information, however, they change the representation of the information.

	Automation	Guidance
Analysis	Constraint checker, model analyzer	Code reviewer, test case designer
Synthesis	Scheduler	Modeling tool
Derivation	Code generator	Guided transformation

Fig. 1. Process activity types

A summary of these categories are shown in Fig. 1. It is important to stress that the mode of execution and design information handling aspects are practically orthogonal, so relevant examples for all combinations can be easily given (in the table cells in Fig. 1).

2.2 Synthesis

Synthesis activities are typically carried out by off-the-shelf development tools such as programming environments, documentation tools, modeling editors etc. By these means, engineers create and manipulate the artefacts of the design process using manual or semi-automated procedures.

Along with design information, most (critical and expensive) *errors* (e.g. design flaws, anti-patterns) are introduced into the system-under-design in these activities. To reduce the impact of these faults, advanced tools offer checking facilities, ranging from light-weight approaches (such as syntax analysis for source code, domain-specific well-formedness checking in model editors) to more advanced features (e.g. static code analysis, model simulation by in-place execution [11]).

The quality of design synthesis output can also be improved by using semi-automated tools for design-time optimization. For instance, in embedded architecture design, automatic *schedulers* may be used to calculate the timing of a message bus communication protocol, or *resource allocation tools* may be used to optimally assign software tasks to hardware nodes.

2.3 Analysis

Figure 2 shows a typical approach to early analysis using model-based techniques. In model-driven development, system requirements and design are captured by high-level, visual engineering models (using popular and standardized modeling languages like UML, SysML, AADL). In critical systems, where the system under design must conform to high quality and reliability standards, early systematic formal analysis of design models plays an increasingly important role to reveal design flaws as early as possible. In general, this can be carried out by generating appropriate mathematical models by automated model transformations. Formal analysis then retrieves a list of problems, which can be fixed by refinement corrections prior to investing in manual coding for implementation. Finally, these refined models may serve as input to code generators and deployment configuration generation, which create the runnable source code of the

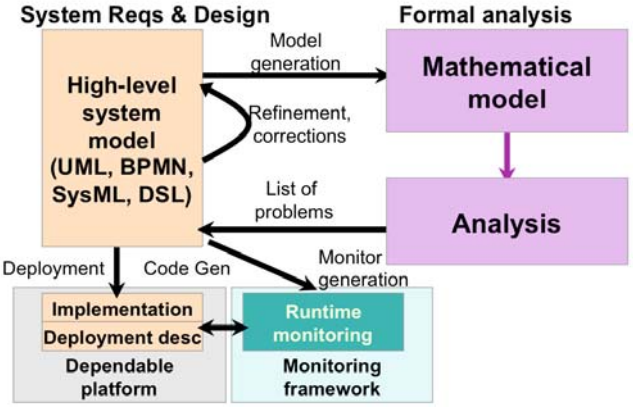


Fig. 2. Analysis by formal techniques

application as well as parametrize auxiliary deployment infrastructure services such as monitoring.

Analysis steps may include (i) to investigate functional correctness of the design by verifying safety or liveness properties (e.g. by model checking statecharts [12,13]), (ii) to assess the effects of error propagation (e.g. using fault modeling and analysis techniques [14]), (iii) to evaluate non-functional characteristics of the system such as performance, availability, reliability or security (e.g. by deriving stochastic models from engineering models [15,16,17]) and many more. A commonality in these approaches is the extensive use of automated model transformations to carry out the abstraction necessitated by the formal analysis.

2.4 Derivation

Derivation steps primarily include automated code generation tasks, or to chain up several development steps by importing and exporting models in different tools.

Derivation steps can frequently be fully automated as all information required for code generation is available prior to initiating the step. Of course, such code generators may still combine the information embedded in different models to derive the designated output, or to mark the platform independent models by design decisions. Anyhow, in both cases, the actual derivation step is completed using existing information implanted by synthesis steps.

Code generators may derive the source code or the target application (see code generators of statecharts [18]), yield deployment descriptors for the target reliable platform [19,20], or generate runtime monitors [21].

As a summary, complex tool integration frameworks should be closely aligned with development processes by taking a transformation-based view on individual development steps. Moreover, they need to simultaneously provide support to integrated automated as well as interactive, user-guided development steps where the starting point and the completion of each step needs to be precisely identified. Finally, the framework

should enable to integrate arbitrary kind of development steps including synthesis, analysis and derivation tasks.

3 Process-Driven Tool Integration

Based on the experience in tool integration summarized in Sec. 2, we propose an integrated approach for designing and executing tool integration processes for model driven development. As a proof-of-concept, we describe a case study developed for the DIANA [6] research project (Sections 4, 5 and 6).

By our approach, development processes are formally captured by workflow models, which (i) specify the temporal macro structure of the process as a task-oriented, hierarchic workflow model; (ii) precisely map the steps of the process to the development infrastructure, consisting of human resources (roles), tools, supporting technologies available as services and development artefacts as entities in the data repository; (iii) define high-level contracts to each step, which specify constraints that help to verify and trace the correctness of outcomes and activities.

These process models are deployed to a software infrastructure, which serves as an automated execution and monitoring platform for the development process. It provides support for running automated and user-guided activities in a distributed environment consisting of heterogeneous software tools, and a model bus-like [22] data repository.

In this section, we describe the specification language of the workflow models in the tool integration domain in detail.

3.1 Process Metamodel

Macro structure. For the specification of the temporal macro structure of development processes, we follow the notions and concepts of well known process description languages such as BPMN or XPDL. As the metamodel in Fig. 3 shows, processes are constructed from workflow `steps` (corresponding to distinct activities carried out during development), and may use standard control flow features such as sequences (`ProcessNode.next`), concurrency (fork-join) and decision points.

More advanced constructs, such as waiting states are intentionally omitted from this language, since this is intended to be a very high level description, where only the order (precedence or concurrence) of activities is important; for execution, this language is mapped to a lower level jPDL representation which may be customized and augmented with more advanced behavioral properties.

Hierarchy. It is important to stress the *hierarchical* nature of the process description: through the `Step.subNodes` relation, workflow steps may be embedded into each other to create a hierarchical breakdown. This allows the process designer to map the ”birds-eye-view” structure development processes (such as phases and iterations) to our language; additionally, it supports ”drill-up-drill-down”-style navigation through a complicated workflow, which is important to reduce design complexity for large-scale processes.

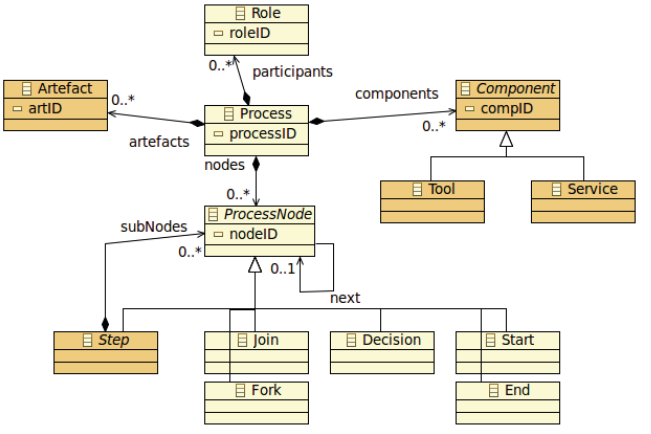


Fig. 3. Integration process macro structure metamodel

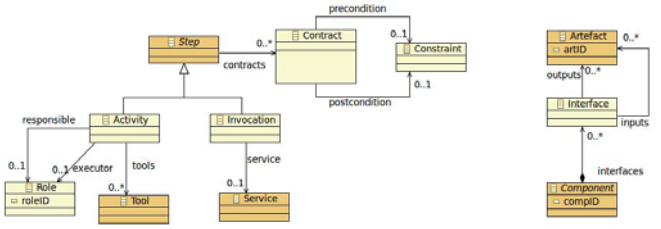
Development infrastructure. Besides the causality breakdown, the language features the following notions of the development infrastructure:

- `Artefacts` represent data structures of the process (e.g. documentation, models, code, generated files, metadata, traceability records).
- `Roles` correspond to the participants of the process, humans and external entities who are involved in executing the user-guided activities.
- `Components` are either `Tools` or `Services` which are used by the participants or invoked automatically during development, to complete a specific step.

These concepts enable the precise mapping of the development workflow to the actual execution infrastructure.

Mapping to the execution infrastructure. As shown in Fig. 4, our language can be used to precisely specify the execution of the development process. During process modeling, workflow steps are first assigned into the `Activity` or `Invocation` categories, depending on the type of interaction (activities are user-guided while invocations are automated). For user-guided tasks, the language adopts the basic role-based assignment method (taken from BPMN), and optionally supports two basic types of relations (`responsible` and `executor`) to indicate which group of users may supervise and actually carry out a task.

Activities may make use of `Tools`, while *invocations* refer to `Services`. From the workflow system's perspective, both tools and services are external software components, which are accessible through interface adaptors. These are represented in the language as `Interfaces` (Fig. 4(b)), which may be connected to artefacts to indicate data flow (`input/output`). At run-time, both tools and services shall be triggered by the process execution engine, with parameters referring to data repository automatically supplied, so that the user does not have to care about managing data and files.



(a) Steps and contracts, services, tools (b) Component-artefact integration

Fig. 4. Integration process metamodel: auxiliary elements

Contracts. In order to guarantee that the result of a step is acceptable and the process can continue, the definition of *contracts* [23] is a well known paradigm. The idea is to *guard* both the input and output of a step by specific constraints. Thus, a `contract` is composed of a `precondition` and a `postcondition`. A precondition defines constraints that needs to be fulfilled by the input of the step in order to allow its execution, while the postcondition guarantees that the process can continue only if its constraints are satisfied by the output. This general definition of a contract allows to use arbitrary formalism (e.g., OCL, JML, etc.) to capture the pre- and postconditions, but could require further refinement of the metamodel with the additional elements of the used formalism.

A detailed example of configuration generation using our graph pattern based contract definition in the context of avionics domain (as part of our case study) can be found in Sec. 4.2.

4 Case Study: The DIANA Toolchain

DIANA (***D**istributed, equipment-**I**ndependent environment for **A**dvanced avio**N**ics **A**pplications* [6]) is an aeronautical research and development project. It aims at the definition of an advanced avionics platform named AIDA (Architecture for Independent Distributed Avionics), supporting (i) execution of object-oriented applications over virtual machines [24], (ii) high-level publish subscribe based communication, and (iii) the applicability of model driven system development (MDSD) in the avionics development domain.

The DIANA project aims to create an MDSD based tool chain for the analysis and generation of ARINC653 [25] real-time operating system (RTOS) configuration files from high-level specifications. Transforming these high-level models into RTOS-specific configuration artefacts is a complex task, which needs to bridge a large abstraction gap by integrating various tools. Moreover, critical design decisions are also made at this stage. For this reason, the use of intermediate domain specific models is advantageous to subdivide the process into well-defined steps and precisely define the interactions and interfaces among the tools used.

In order to introduce the DIANA approach Section 4.1 focuses on the models and metamodels used through the mapping process, while Section 4.2 gives an overview on the actual steps of the workflow.

4.1 Models

In the DIANA project the aim of the high-level Platform Independent Model (PIM) is to capture the high-level architectural view of the system along with the definition of the underlying implementation platform, while the Platform Specific Model (PSM) focuses on the communication details and service descriptions.

Platform Independent Models. In order to support already existing modeling tools and languages (e.g., Matlab Simulink, SysML, etc.) we use a common architecture description language called *Platform Independent Architecture Description Language* (PIADL) for architectural details by extracting relevant information from supported common off-the-shelf models. As for capturing the underlying platform (in our case ARINC653) we use a *Platform Description* model (PD) capable of describing common resource elements.

- PIADL aims to provide a platform independent architectural-level description of event-based and time-triggered embedded systems using message and publish/subscribe based communication between jobs, having roots in the PIM metamodel of the DECOS research project [26].
- The *Platform Description* (model) describes the resource building blocks, which are available in an AIDA Module to assemble the overall resources of an AIDA component. This mainly includes ARINC653 based elements such as modules, partitions, communication channels, etc. A small part of the metamodel is detailed in Section 5.2.
- In the context of the DIANA project we support *Matlab Simulink* as a source COTS language. We supports only a fraction of the language that conforms with the expressiveness of our PIADL to describe the high-level architecture of the system.

Platform Specific Models. The platform specific models are encapsulated in the *AIDA Integrated System Model* that contains all relevant low-level details of the modelled system. Essentially based on ARINC653, the integrated model provides extensions and exclusions to support the publish/subscribe communication and service based invocations. Its main parts are the following:

- The *Interface Control Document* (ICD) is used to describe data structures and low-level data representation of AIDA systems, interfaces and services to ease integration of the described element with other parts of the system. It supports both high-level (logical) and low-level (decoding) descriptions and was designed to be compatible with the ARINC653 and ARINC825 data and application interface descriptions.

– The *AIDA System Architecture* model identifying and describing the relations among all elements related to the AIDA system. More precisely the model focuses on the (i) details of the proposed publish/subscribe based communication, (ii) the multi-static configuration of the AIDA middleware and (iii) the detailed inner description of the partitions allocated for the AIDA system.

In order to support traceability – an essential requirement of DO-178B [1] certification –, a trace element is saved in the *Trace* model for all model elements of the PSM created during the mapping process. Such an element saves all PIM model segments that were used for the creation of a PSM model element. Additionally, trace information is also serialized into separate XMI files for each generated configuration file. In the current implementation traceability is hand-coded separately into each step of the development workflow.

4.2 Overview of the DIANA System Modeling Process

An extract of the defined workflow for the *DIANA System modeling* process is depicted in Figure 5, using a graphical concrete syntax of the process metamodel presented in Figure 3 and Figure 4.

The process starts with the definition of a `complete PIADL model` as the task of the `System architect` (represented by a human symbol). It can be either manually

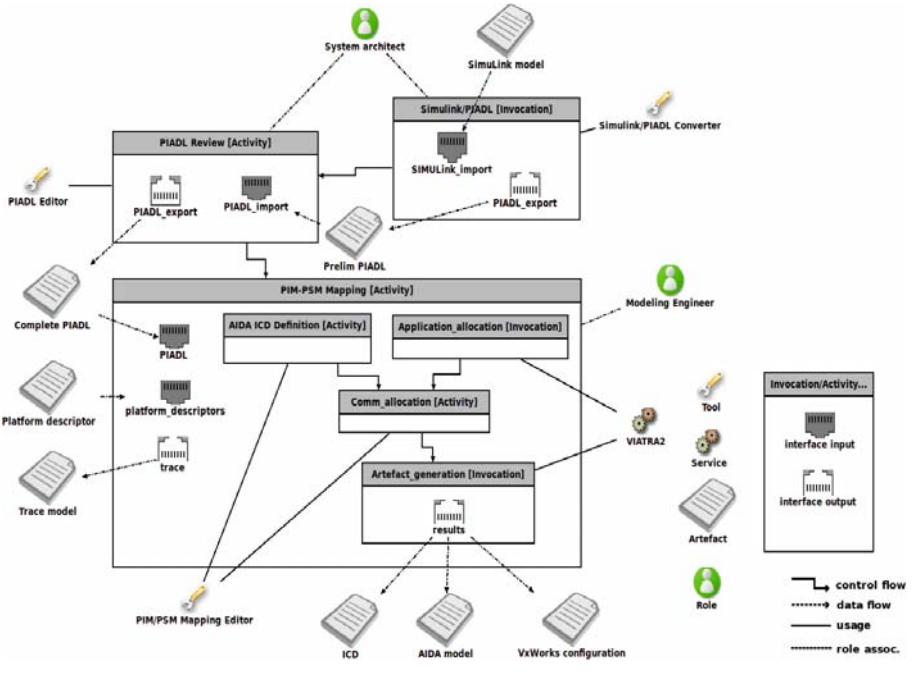


Fig. 5. Overview of the DIANA development process

defined using the (i) external PIADL editor (depicted by a wrench icon) as part of the PIADL review step or (ii) derived from a Simulink model.

The near one-to-one derivation is supported by the Simulink/PIADL Converter external tool used in the PIADL Review step. It has an input and an output interface figured by a grey and white socket symbol for the Simulink and the PIADL model, respectively. However, as some AIDA specific parameters cannot be directly derived it requires additional clarification from the system architect. For example, a *subsystem* block in the Simulink model is mapped to a *job* in the PIADL, but its modular redundancy value (how many instances of the job are required) is not present in the Simulink model.

The complete PIADL is then imported into the PIM/PSM mapping editor responsible for the analysis and definition of configuration tables interface descriptions. This work is done by the Modeling Engineer. Without going into more details it consists of 25 steps organized into the following main categories:

1. *Application allocation*: contains the PIM imports followed by the allocation of application instances to partitions and steps that define additional constraints on the allocation. It relies on the VIATRA2 framework and depicted by an invocation step.
2. *AIDA ICD definition*: steps related to the description of interfaces and services provided and required by applications. These are user driven mapping steps, where PIM types, messages, topics and services are refined with platform specific information like encoding, default value, etc. It is supported by the PIM/PSM mapping editor.
3. *Communication allocation*: involves steps in the PIM/PSM Mapping editor that carry out the allocation of inter-partition communication channels and the specification of ports residing on each end of these channels.
4. *Artefact generation*: contains steps that carry out the generation of AIDA middleware model, ARINC653 configuration files for the VxWorks real-time OS and the AIDA ICD descriptor.

Additionally, as a cross cutting aspect traceability information - depicted by the Trace model - is saved during the mapping process.

5 Graph Transformation in the DIANA Tool Chain

This section gives an overview how we successfully applied graph transformation based technology in various parts of the tool chain. Section 5.1 introduces a graph pattern based contract notation used to define conditions for steps, along with an example detailed in Section 5.2. Section 5.3 highlights how graph transformation is used for ARINC configuration generation.

5.1 Contracts as Graph Patterns

During a development process certain steps require external COTS tools (e.g., Matlab, SAL, etc.) or user interaction to perform their task. As mentioned in Section 3.1 we use

contracts to ensure that both the input and output of these steps satisfy their requirements. In our approach we used *graph patterns* to capture such contracts [27, 28] as we used on-the-fly evaluation based on incremental pattern matching [29]. However, it is important to note, that it would be possible to define these contracts by OCL and transform a large subset directly to the graph pattern formalism [30].

Graph patterns are frequently considered as the atomic units of model transformations [31]. They represent conditions (or constraints) that have to be satisfied by a part of the underlying instance model. In the following, we use the pattern language of the VIATRA2 framework [32]. In our interpretation a basic graph pattern consists of graph elements corresponding to the metamodel. As an addition for more complex pattern specification the language of VIATRA2 allows to define *alternate* (OR) *pattern bodies* for a pattern, with a meaning that the pattern is fulfilled if at least one of its bodies is fulfilled. A *negative application condition* (NAC) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [33].

Contracts are composed of a pre- and a postcondition. Both conditions are the conjunction of subconditions described by graph patterns, where a graph pattern is a disjunction of alternate pattern bodies [31]. A subcondition described by the graph pattern is either a *positive* or *negative* condition. A negative condition is satisfied by a model if it does not have a match in the underlying model. While a positive one is satisfied if its representing graph pattern has a match in the model. A further restriction on positive condition can be formulated by stating that they are satisfied iff their representing graph pattern has a *predefined* positive number (*Cardinality*) of matches.

5.2 Example

To demonstrate how contracts can be formulated using the defined approach consider the simplified *job allocation* step of the *Application Allocation* category (See in Section 4.2) using an external tool (the VIATRA2 framework). In this step the task is to allocate an IMA system defined by its jobs and partitions over a predefined cabinet structure and to minimize the number of *modules* used. An integrated modular avionics (IMA) system is composed of *Jobs* (also referred as applications), *Partitions*, *Modules* and *Cabinets*. *Jobs* are the atomic software blocks of the system defined by their memory requirement. Based on their criticality level, jobs are separated into two sets: *critical* and *simple* (non-critical). For critical jobs, double or triple modular redundancy is applied while for simple ones only one instance is allowed. *Partitions* are complex software components composed of jobs with a predefined free memory space. Jobs can be allocated to the partition as long as they fit into its memory space. *Modules* are SW components capable of hosting partitions. Finally, *Cabinets* are HW storages for maximum (in our example) two modules used to physically distribute elements of the system. Additionally a certain number of safety related requirements will also have to be satisfied: (i) a partition can only host jobs of one criticality level and (ii) instances of a

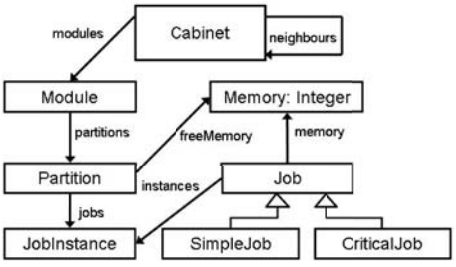


Fig. 6. Metamodel of an IMA architecture

certain critical job can not be allocated to the same partition. An excerpt of the Platform Description metamodel describing the detailed IMA system is depicted in Figure 6.

Based on this metamodel we defined the pre- and the postcondition of this step as depicted in Figure 7 and Figure 8, respectively. All subconditions in the pre- and the postcondition are defined as *positive* and *negative* conditions depicted with with + and - markings, respectively.

Precondition. For the definition of the precondition we rely only on that the model has at least one cabinet, one partition with its free memory defined and one job with an instance. These simple requirements are captured by the *cabinet*, *partition* and *job* graph patterns depicted in Figure 7.

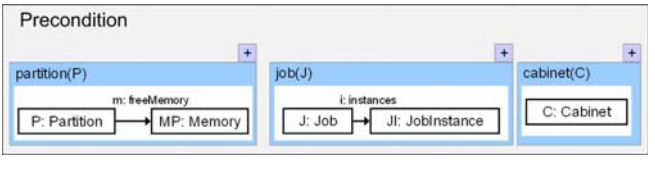


Fig. 7. Precondition of the DIANA application allocation step

Postcondition. The *jobInstancewithoutPartition*, *partitionwithoutModule* and *modulewithoutCabinet* subconditions describe that in a solution model each *JobInstance*, *Partition* and *Module* is allocated to a corresponding *Partition*, *Module* and *Cabinet*, respectively. For example, the *jobInstancewithoutPartition* subgoal captures its requirement using a double negation (NAC and negative constraint) stating that there is *no unallocated* job instance *JI* in the solution model. As the declarative graph pattern formalism has an implicit existential quantification, nested (double) negation is required to quantify elements universally. Similar double negation is used in case of the other two subgoals.

The rest formulates safety and memory requirements. The *partitionMemoryHigherThan0* pattern captures the simple memory constraint that all partitions must have higher than zero free memory. The safety requirement stating that a partition can only

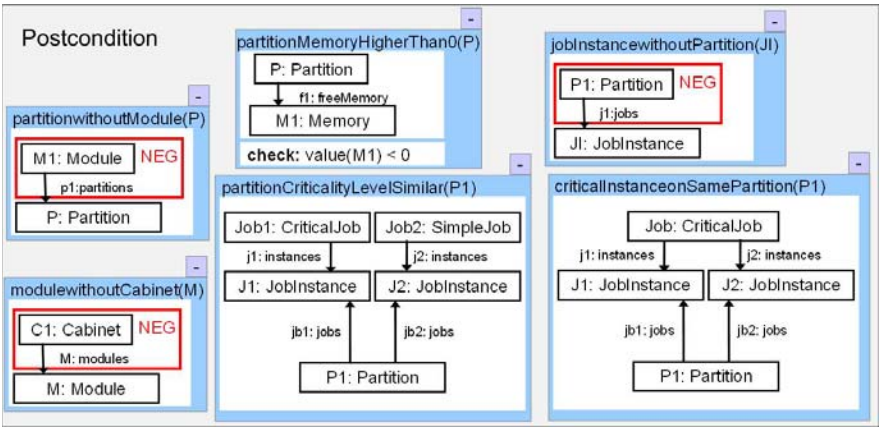


Fig. 8. Postcondition of the DIANA application allocation step

host jobs of one criticality level is captured by the *partitionCriticalityLevelSimilar* pattern. As it is a *negative constraint* it describes the (positive) case where the *P1* partition holds two job instances *J1* and *J2* of a simple and a critical job *Job1* and *Job2*, respectively. The *criticalInstanceonSamePartition* and *criticalInstanceonSameModule* patterns restrict in a similar way that no job instances *J1* and *J2* of a critical job *Job* can be allocated to the same partition *P1* or module *M1*.

Guarding the application allocation using this contract ensures that all applications are properly allocated and all safety requirements are fulfilled by the output model. As constraints are defined by graph patterns it gives rise to adapting constraint satisfaction programming techniques over graph transformation as in [34].

5.3 Configuration Generation by Graph Transformation

Model transformation based automatic code or configuration generation is one of the main driving forces [35,36] of model driven system development. It offers many advantages including the rapid development of high quality code, reduced number of errors injected during development and the consistency between the design and the code is retained, in comparison with a purely manual approach.

One aim of the DIANA project is to generate ARINC653 [25] XML based configuration files for VxWorks 653 RTOS from the created PSMs. A configuration file describes the internal structure of a module, namely: (i) allocated partitions and their memory layout, (ii) communication channels over sampling and queueing ports and (iv) health monitor tables for error detection and handling. During the PIM-PSM mapping process all relevant information required for the generation are specified and stored in the AIDA Integrated System model.

An example ARINC653 configuration snipplet is depicted in Figure 9. It captures the details of the *flight management non-system* partition, which has the highest *Level A* criticality as defined in [1], one queueing and four sampling ports and separate memory blocks for *code* and *data*. A port is defined with its *direction*, *maximum message size*

and *name*, where the sampling and the queuing ports have additionally *refresh rate* or *maximum number of messages* parameters, respectively. Finally, a memory block is defined by its *access* mode (e.g, read or write), *type* (code or data) and *size*.

```
- <Partition Criticality="LEVEL_A" EntryPoint="initial" PartitionIdentifier="p187" PartitionName="flight management" SystemPartition="false">
    <Sampling_Port Direction="DESTINATION" MaxMessageSize="1088" Name="ACP_1_TO_PFC_1_AND_PFC_2_R1" RefreshRateSeconds="0.25" />
    <Sampling_Port Direction="DESTINATION" MaxMessageSize="192" Name="PP_1_TO_SD_1___C0_R0" RefreshRateSeconds="0.1" />
    <Sampling_Port Direction="DESTINATION" MaxMessageSize="576" Name="ZC_1_TO_PTC_1_AND_PTC_2_R0" RefreshRateSeconds="1.0" />
    <Sampling_Port Direction="DESTINATION" MaxMessageSize="576" Name="ZTP_1_TO_ZC_1_AND_SD_1_R1" RefreshRateSeconds="0.5" />
    <Queuing_Port Direction="DESTINATION" MaxMessageSize="64" Name="ACP_1_TO_PTC_1_AND_PTC_2_R0" MaxNbMessages="128" />
  </Partition>
+ <Partition Criticality="LEVEL_A" EntryPoint="initial" PartitionIdentifier="p939" PartitionName="IO processing" SystemPartition="true">
+ <Partition Criticality="LEVEL_A" EntryPoint="-" PartitionIdentifier="___mw821" PartitionName="Middleware_for_Channels" SystemPartition="true">
+ <Partition_Memory PartitionIdentifier="p996" PartitionName="flight controls">
- <Partition_Memory PartitionIdentifier="p187" PartitionName="flight management">
    <Memory_Requirements Access="READ_WRITE" SizeBytes="65536" Type="CODE" />
    <Memory_Requirements Access="READ_WRITE" SizeBytes="32768" Type="DATA" />
  </Partition_Memory>
```

Fig. 9. Example ARINC653 descriptor

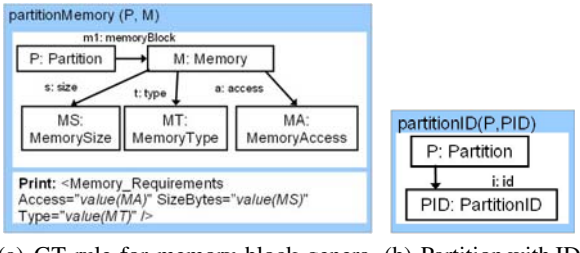
Configuration Generation. To generate the required XML format we based our code generator on the VIATRA2 framework, where configuration file templates are defined using graph transformation (GT) rules. This way GT rules define both the GT pattern that matches to the required elements in the model and generate code as a side effect. These GT rules do not modify the underlying model and thus their right and left hand sides are the same.

Without going into details, a part of the code generator responsible for the generation of the Partition_Memory XML subtree is depicted in Figure 10 and Listing 1.1.

The *partitionMemory* GT rule defines the template for the *Memory_Requirements* XML element. Its pattern matches to the partition *P* that has a memory *M* as its memory block. A memory has three attribute defined as memorySize *MS*, memoryType *MT* and memoryAccess *MA*. The *print* block defines the template that is printed out when the rule is applied. All three parameters value are retrieved using the *value* keyword.

The *partitionID* is an auxiliary pattern used to get the ID *PID* of the partition *P*.

To control the execution of the GT rules and define complex generators the VIATRA2 language [32] uses abstract state machines ASM [37]. ASMs provide complex model transformations with all the necessary control structures including the sequencing



(a) GT rule for memory block genera- (b) Partition with ID tion

Fig. 10. Example GT patterns and rules used for configuration generation

operator (*seq*), ASM rule invocation (*call*), variable declarations and updates (*let* and *update* constructs), *if-then-else* structures, non-deterministically selecting (*random*) constructs, iterative execution (applying a rule as long as possible (ALAP) *iterate*), the simultaneous rule application at all possible matches (locations) (*forall*) and single rule application on a single matching (*choose*).

The example code shown in Listing. 1.1 demonstrates how we defined our code generator using the *partitionMemory* rule and the *partitionID* pattern.

The outer *forall* rule is used to find all partitons *P* with their id *PID* in the model as defined by the *partitionID* pattern, and then execute its inner sequence on all matches separately. For each partition separate *Partition_Memory* XML elements are emitted out with their additional *PartitionIdentifier* and *PartitionName* parameters. As for the *Memory_Requirements* XML elements a *forall* rule invoking the *partitionMemory* GT rule is defined. The rule is invoked for all memory blocks *M* of partition *P*, where *P* is (at that point) already bound to a concrete partition by the outer forall.

The whole code generator is built up using similar snipplets.

```
... //memory block generation
forall P, PID with find partitonID(P, PID) do seq{
  println("<Partition_Memory PartitionIdentifier=\"+value(PID)
          +\" PartitionName=\"+name(P)+\">");
  forall M with apply partitionMemory(P,M); // GT rule as template
  println("</Partition_Memory>");
}
...
```

Listing 1.1. Partition_Memory code generator snipplet

6 Implementation of the Tool Integration Framework

To support the application of the high-level process modeling language presented in Sec. 3, we have created a prototype implementation for the tool integration framework. This framework provides the software infrastructure on which the case study of Sections 4 and 5 is executed. A main design goal was to integrate our solution to existing off-the-shelf tools that are used in industry practice; thus, both the process modeling infrastructure, as well as the execution environment rely on standard technologies as much as possible.

6.1 Execution Architecture

The execution of the process is facilitated by a service-oriented architecture, based on the jBoss jBPM [38] workflow execution engine and the Rational Jazz platform [39], as an integration middleware between tools, services, and the data repository. Building on this software environment, we have implemented a lightweight API that provides essential components, the overall architecture is shown in Fig. 11.

Tool management. A *Tool* or *Service* represents an external executable program that performs one or more tasks during the development. In order to be easily integrated, especially in the case of services, these software components should ideally be programmatically invocable, i.e., have the business functionality exposed to a well-defined

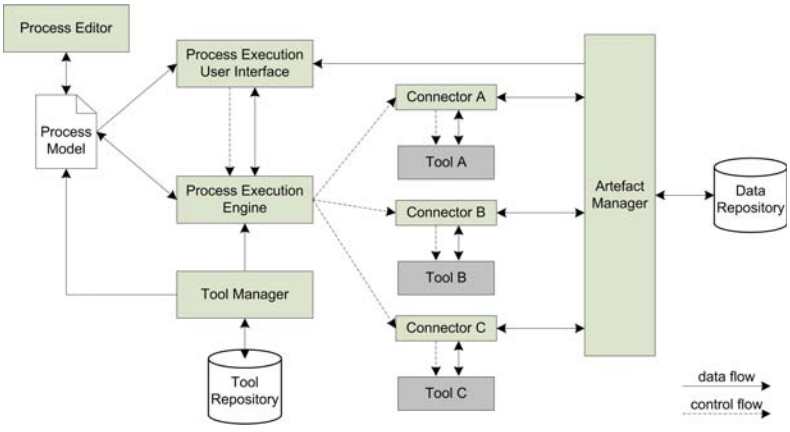


Fig. 11. Framework architecture

interface which is externally accessible (ranging from command line interfaces to library functions or even web services).

Connectors. Connectors are the components that provide uniform interface of the tools for the framework. The connector is also responsible for facilitating data flow between the tool and the artefact repository (optionally, support for explicit traceability may also be implemented in the Connector). The *Tool Manager* together with the *Tool Repository* serve as a service directory for available tools and services. It relies on the underlying service facilities of the Rational Jazz and OSGi platforms. These components are responsible for the lifecycle management (initialization, disposal) of integrated tools.

Data management. Models and artefacts of the development process are described (on a high abstraction level) in the Process Model. From this, a *storage metamodel* is generated, which contains dependency references between artefact classes, and includes metadata (such as creation timestamps, ownership and access control flags) as attributes. *Traceability* information is also handled as storage metamodel classes. The *Artefact Manager* is responsible for providing access through a service-oriented API (implemented as a tool/service interface) to data records stored in the *Data Repository* component.

Process execution. The executing processes can be managed and supervised using the *Process Execution User Interface*. In our prototypical implementation, it provides a control panel where (i) execution of tasks defined in the platform-specific process model can be initiated, (ii) the state of execution (i.e. the current process node, and the process variable values) can be observed, and (iii) versions of artefact instances and their related metadata can be managed.

The *Process Execution Engine* is responsible for the execution of the steps defined in the Process Model. The process model is mapped to a low-level executable language (jBoss jPDL [38]), which is executed in a customized jBPM instance. The jPDL description contains auxiliary information that is processed by *handler plugins*, so that

the process executor is able to invoke the integrated tools, services, and access the data repository.

6.2 Process Modeling Languages for Tool Integration

In practice, the domain-specific language of Sec. 3 is not the only means of designing a development process; in fact, several modeling languages may be involved on two levels of abstraction (Fig. 12).

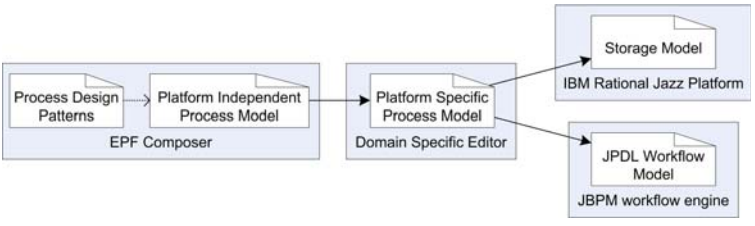


Fig. 12. Models and their relations used in process modeling and execution

High-level process models. In today's industrial practice, development processes are frequently captured in process description languages with a focus on methodology-compliance (i.e. enforcing design principles so that the actual development conforms to standard methods such as the Unified Process, or modern agile approaches such as XP or SCRUM). To address this need from a metamodeling perspective, the Software Process Engineering Metamodel (SPEM) [40] has been developed by the OMG. Since then, a number of SPEM-based tools have emerged, and IBM Rational's Method Composer is one of the most well-known of them. Along with its open-source version, the Eclipse Process Framework Composer [41] (shown in Fig. 13), they are based on pattern re-use by allowing to design according to process libraries that incorporate deep knowledge of both standard methodologies (e.g. OpenUP) and also organization-specific customizations.

Presentation Name	Index	Predecessors
DIANA System Modeling Process	0	
Matlab Modeling	1	
PIM-PSM Modeling	2	
Simulink-PIADL Conversion	3	
PIADL Review	4	3
PIM-PSM Mapping	5	4
AIDA ICD Definition	6	
Application Allocation	7	
Communication Allocation	8	6,7
Artefact Generation	9	8

Fig. 13. The DIANA process in the EPF Composer

As EPF's language includes support for the high level enumeration of roles and artefacts, with lightweight associations (such as responsibility, input-output), a "platform-independent" representation of development processes may be designed. Note that in

these models, all activities appear as *tasks* (Fig. 13), so there is no information present about which elements are used-guided and which are automated.

Thus, this high level model can mapped to our DSML by a VIATRA2 transformation, preserving the macro structure of the process, and importing an enumeration of roles, tools and artefacts. This domain-specific model has to be augmented manually to precisely specify how activities interact with artefacts, tools, services, and their interfaces.

Storage models. Two types of deployment models are generated from the platform-specific model: (i) the *Workflow Model* contains the description of the tool-chain to be executed in the format (augmented jPDL) that can be executed by the Process Execution Engine, and (ii) the *Storage Model*, which is the description of the data structure in a format that is needed to configure the Rational Jazz Data Repository.

In Jazz, *storage models* [42] are EMF/Ecore-compliant metamodels that define an object-oriented database schema in which artefacts can be stored. Inter-class references indicate cross-references between repository elements (for instance, such cross references may be used to determine which document instances need to be manipulated synchronously to maintain consistency).

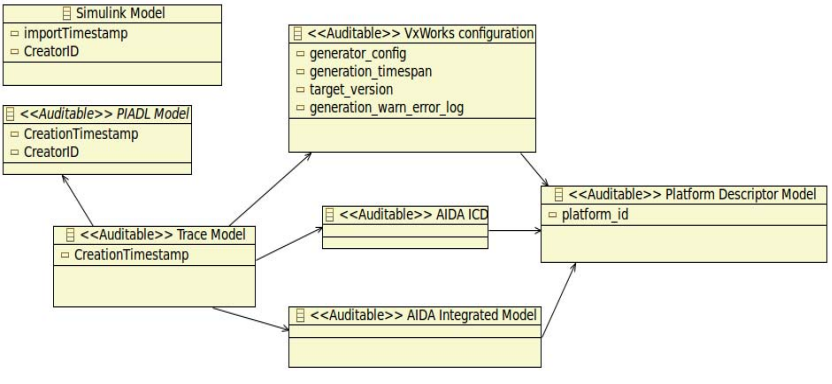


Fig. 14. Storage metamodel for the DIANA process

A sample storage model extract for the DIANA case study is shown in Fig. 14. Classes, tagged with the *Auditable* stereotype, are under versioning persistence management, and can store metadata as attributes. These attributes can be queried and processed without retrieving the artefacts in their entirety. Note that in this example, we do not record any traceability information between the Simulink model and the rest of artefacts, hence it is shown as a separate auditable entity in the storage metamodel.

Metadata attributes typically include lightweight traceability information (e.g. creation timestamps, creator IDs which may refer to a particular user or the ID of an automated service), and may also incorporate logs and traces (such as, for instance, the *generation_warn_error_log* attribute for the VxWorks configuration document, which contains the warning and error log emitted by the code generator during generation).

These auxiliary records, together with a complete *trace model* (also represented as a persistent and versioned artefact) play an important role in achieving *end-to-end traceability*.

Based on this storage model, a persistence plug-in is generated for the Jazz repository, which allows the tool connectors (as well as external components) to interact with the database on the artefact level. This interface currently only provides basic access functionality (queries and manipulations):

- getter functions for all document types and their metadata (e.g.`getVxWorks_configurations()`, `.getGenerator_config()` etc.), which directly retrieve data records from the artefact database and wrap them into EMF objects;
- getter functions for complex queries involving storage classes with cross references (e.g. `getTrace_model_transitive(String traceModelId)`, which fetches a particular trace model document together with all its referenced document instances);
- manipulation (setter) functions for all document types and their metadata attributes, which take EMF objects as data transfer parameters (e.g. `storeVxWorks_configuration(VxWorksConfigurationModel)`).

7 Related Work

The problem of tool integration has already been studied in many different research projects whose relationships to our proposed approach are now surveyed.

The UniForM WorkBench [43] can be considered as one of the earliest attempts for tool integration due to its built-in support for type safe communication between different tools, version and configuration management. Though system models can be interchanged in a type safe manner by the workbench, it cannot be considered as a model-based approach as a whole.

Several technology dependent approaches have already been proposed for tool integration purposes. One valuable representative of this group is R-OSGi [44], which supports the deployment of distributed applications on computers having the OSGi framework installed. Though the underlying OSGi framework has many advanced services, the centralized management (i.e., loading and unloading) of modules is an inconvenient property of R-OSGi. Another representative is the jETI system [45], which is a result of redesign and Java-based reimplementation of the Electronic Tool Integration platform, is an approach based on the Eclipse Plugin architecture whose technology dependency has been reduced by its Web Services support. The jABC submodule of the jETI system enhances Java development environments with remote component execution, high-level graphical coordination and dedicated control via formal methods.

The use of workflows for describing the tool integration process, which is a technique also employed in our approach, has been introduced in the bioinformatics domain in [46]. In this paper, the authors proposed to describe the cooperation of computational tools and data management modules by workflows.

The first form of metamodel-based tool integration appears in [22], which presents two orthogonal design patterns as well. The first pattern suggests the storage of metadata on a server, and the development of a model bus, on which tools can transfer models via a common model interface protocol. The other pattern proposes the use of workflows for describing the tool integration process in the ESML language.

Model transformations in tool integration. In the followings, tool integration solutions with model transformation support are presented.

In the authors' experience, VIATRA2, positioned as a dedicated model transformer, has been successfully applied both in scenarios where the abstraction gap (between source and target languages) was relatively small (such as code generation from MDA-style platform-specific models [19,47,48,49], or abstract-concrete syntax synchronization in domain-specific languages [50]), as well as mappings with strong abstractions (e.g., the generation of mathematical analysis models from design artefacts, for formal analysis purposes).

The IPSEN approach [51] outlined probably the first integration related scenario, where model transformation techniques played a key role. The aim of IPSEN was to construct an integrated software development environment (SDE) tool, which helped capturing both context-free (i.e., syntactic) and context-sensitive (i.e., graph-based) aspects of languages by textual and graphical editors, respectively. The technique of graph transformation has been heavily used for the development of the tool especially for specifying constraints and translations in the context-sensitive domain.

ModelCVS [52] employs (i) semantic technologies in forms of ontologies to partly automate the integration process, and (ii) QVT transformations, which are generated from these ontology descriptions. As distinctive features, ModelCVS uses Subversion for versioning, EMF and MOF-based metamodels for model representation, and a generic workflow ontology for defining processes. In contrast to our approach, ModelCVS prepares adapters for tools and not for models as these latters are stored in a central repository. Additionally, model transformations are used in ModelCVS for the synchronization of models, and not for the definition of the integration process.

From the model transformation point of view, a similar setup can be found in MOFLON [53,54]. Transformations are again used for model synchronization, but in this case, they are defined by triple graph grammars. MOFLON operates on JMI and MOF 2.0 based models.

TopCased ("The Open source toolkit for Critical Systems") [55] is a software environment primarily dedicated to the realization of critical embedded systems including hardware and/or software. Topcased promotes model-driven engineering and formal methods as key technologies, such as a model bus-based architecture supporting standard modeling technologies such as EMF, AADL, UML-MARTE, and SysML. For model transformations, TopCased uses ATL [56].

The recent EU projects of ModelWare [57] and MODELPLEX [58] outline techniques that show certain similarity to our approach. ModelWare aimed at defining and developing the complete infrastructure required for large-scale deployment of MDD strategies and validating it in several business domains. It can (i) provide transparent integration across model, tool, platform, machine boundaries; (ii) support the creation of distributed, multi-user tool chains; (iii) handle many metamodels and artefacts; (iv) integrate interactive and non-interactive tools; and (v) use different technologies for communication. ModelWare offers a process modeling framework, and a model bus for exchanging high-level data that are either Java-based or described by Web Services. On the other hand, it lacks model transformation support, which has only been added in its successor MODELPLEX project. MODELPLEX has a SPEM2 based toolset for

supporting the enactment and execution of processes and is integratable with workflow and project management tools as well.

The clear separation of PIMs and PSMs, which specify tool integration processes with different levels of details can only be found in research projects GENESYS [9] and DECOS [5], which propose a cross-domain architecture for embedded systems, and a model-driven development process for avionics systems, respectively. As distinctive features, GENESYS supports (i) different modeling languages including UML and many of its profiles, (ii) a service-oriented development of subsystems, (iii) both uni- and bidirectional model transformations with manual, semi-automatic, automatic execution.

8 Conclusion

In the paper, we proposed a tool integration framework, which is centered around a high-level domain-specific language for process modeling to closely align tool integration and development scenarios. With this approach, model-driven development processes can be described precisely, in detail that is sufficient to capture what can and should be automated, but also flexible enough to support user-guided steps as well.

Based on our experience in designing and implementing tool chains primarily in the embedded and service-oriented domains, a key issue is to precisely specify and validate the individual elementary steps of the development tool chain. For this purpose, we adapted graph patterns to formally specify contracts for each step. Furthermore, model transformations provided by graph transformation techniques were responsible for fully automating certain steps in the tool chain (like code generation or model analysis tasks).

In addition to the process-driven specification of tool integration chains, we have also presented an execution framework, which is rooted on various research projects at the group. This framework is built to accommodate a wide spectrum of Eclipse-based or external) tools, and automatically execute development processes designed with our modeling language.

As future work, we primarily aim at including support for advanced model bus services, such as versioning, model merge, and automatic traceability information generation. Additionally, we are planning to integrate advanced support for automated traceability based on change-driven transformations introduced in [50].

References

1. RTCA - Radio Technical Commission for Aeronautic: Software Considerations in Airborne Systems and Equipment Certification, DO-178B (1992), https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=633
2. Rushby, J.: Runtime Certification. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 21–35. Springer, Heidelberg (2008)
3. Kornecki, A.J., Zalewski, J.: The Qualification of Software Development Tools from the DO-178B Perspective. Journal of Defense Software Engineering (April 2006), http://www.stsc.hill.af.mil/crosstalk/2006/04/0604KorneckiZalewski.html
4. Miller, S.P.: Certification Issues in Model Based Development Rockwell Collins

5. The DECOS Project: DECOS - Dependable Embedded Components and Systems, http://www.decos.at/
6. The DIANA Project Consortium: DIANA (Distributed, equipment Independent environment for Advanced avioNic Application) EU FP6 Research Project, http://dianaproject.com
7. The SENSORIA Project: The SENSORIA website, http://www.sensoria-ist.eu
8. The MOGENTES Project : MOGENTES (Model-based Generation of Tests for Dependable Embedded Systems) EU FP7 Research Project, http://mogentes.eu
9. The GENESYS Project: GENESYS - GENeric Embedded SYStem, http://www.genesys-platform.eu/
10. The INDEXYS Project: INDEXYS - INDustrial EXploitation of the genesYS cross-domain architecture, http://www.indexys.eu/
11. Ráth, I., Vágó, D., Varró, D.: Design-time Simulation of Domain-specific Models By Incremental Pattern Matching. In: 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (2008)
12. Pintér, G., Majzik, I.: Runtime Verification of Statechart Implementations. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 148–172. Springer, Heidelberg (2005)
13. Sisak, Á., Pintér, G., Majzik, I.: Automated Verification of Complex Behavioral Models Using the SAL Model Checker. In: Tarnai, G., Schnieder, E. (eds.) Formal Methods for Automation and Safety in Railway and Automotive Systems (Proceedings of the FORMS-2008 Conference), Budapest, Hungary, LH́armattan (2008)
14. Partaricza, A.: Systematic generation of dependability cases from functional models. In: Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMAT 2008), Budapest, Hungary (2007)
15. Majzik, I., Pataricza, A., Bondavalli, A.: Stochastic dependability analysis of system architecture based on uml models. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677, pp. 219–244. Springer, Heidelberg (2003)
16. Schoitsch, E., Althammer, E., Eriksson, H., Vinter, J., Gönczy, L., Pataricza, A., Csertán, G.: Validation and Certification of Safety-Critical Embedded Systems - the DECOS Test Bench. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 372–385. Springer, Heidelberg (2006)
17. Balogh, A., Pataricza, A., Ráth, I.: Automated verification and validation of domain specific languages and their applications. In: Proceedings of the 4th World Congress for Software Quality, Bethesda, USA, pp. 1–6 (2009)
18. Pintér, G., Majzik, I.: Model Based Automatic Code Generation for Embedded Systems. In: Proceedings of the Regional Conference on Embedded and Ambient Systems (RCEAS 2007), Budapest, Hungary, pp. 97–106 (2007)
19. Gönczy, L., Ávéd, J., Varró, D.: Model-based Deployment of Web Services to Standards-compliant Middleware. In: Isaias, P., Miguel Baptista Nunes, I.J.M. (eds.) Proc. of the Iadis International Conference on WWW/Internet 2006 (ICWI 2006), Iadis Press (2006)
20. Kövi, A., Varró, D.: An eclipse-based framework for ais service configurations. In: Malek, M., Reitenspieß, M., van Moorsel, A. (eds.) ISAS 2007. LNCS, vol. 4526, pp. 110–126. Springer, Heidelberg (2007)
21. Pintér, G., Majzik, I.: Automatic Generation of Executable Assertions for Runtime Checking Temporal Requirements. In: Dal Cin, M., Bondavalli, A., Suri, N. (eds.) Proceedings of the 9th IEEE International Symposium on High Assurance Systems Engineering (HASE 2005), Heidelberg, Germany, October 12-14, pp. 111–120 (2005)
22. Karsai, G., Lang, A., Neema, S.: Design Patterns for Open Tool Integration. Software and Systems Modeling 4(2), 157–170 (2004)
23. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)

24. Locke, C.D.: Safety critical java™technology. In: JTRES 2006: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems, pp. 95–96. ACM, New York (2006)
25. ARINC - Aeronautical Radio, Incorporated: A653 - Avionics Application Software Standard Interface, https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=633
26. DECOS - Dependabe Embedded Components and Systems consortium : The DECOS Platform Independent Metamodel, public deliverable, http://www.inf.mit.bme.hu/decoscd/deliverables/DECOS_deliv_PIM_Metamodel.pdf
27. Baar, T.: OCL and graph-transformations - A Symbiotic Alliance to Alleviate the Frame Problem. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 20–31. Springer, Heidelberg (2006)
28. Azab, K., Habel, A.: High-level programs and program conditions. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 211–225. Springer, Heidelberg (2008)
29. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT 2008, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (2008)
30. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. Electron. Notes Theor. Comput. Sci. 211, 159–170 (2008)
31. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3), 214–234 (2007)
32. Balogh, A., Varró, D.: Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In: ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006) (2006) (in press)
33. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004)
34. Horváth, Á., Varró, D.: CSP(M): Constraint Satisfaction Programming over Models. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 107–121. Springer, Heidelberg (2009)
35. Hemel, Z., Kats, L.C.L., Visser, E.: Code generation by model transformation. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 183–198. Springer, Heidelberg (2008)
36. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Towards graph transformation based generation of visual editors using eclipse. Electr. Notes Theor. Comput. Sci. 127(4), 127–143 (2005)
37. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
38. Koenig, J.: JBoss jBPM White Paper. Technical report, The JBoss Group / Riseforth.com (2004), http://jbossgroup.com/pdf/jbpm_whitepaper.pdf
39. IBM Rational: Jazz Community Site, http://jazz.net/
40. The Object Management Group: Software Process Engineering Metamodel, version 2.0 (2008), http://www.omg.org/technology/documents/formal/spem.htm
41. The EPF Project: The Eclipse Process Framework website, http://www.eclipse.org/epf/
42. Haumer, P.: Increasing Development Knowledge with Eclipse Process Framework Composer. Eclipse Review (2006), http://haumer.net/rational/publications.html
43. Einar W. Karlsen: The UniForM WorkBench: A Higher Order Tool Integration Framework. Lecture Notes in Computer Science **1641** (1999) 266–280

44. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications Through Software Modularization. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 1–20. Springer, Heidelberg (2007)
45. Margaria, T., Nagel, R., Steffen, B.: jETI: A Tool for Remote Tool Integration. LNCS, vol. 2440, pp. 557–562. Springer, Heidelberg (2005)
46. Corradini, F., Mariani, L., Merelli, E.: An Agent-based Approach for Tool Integration. International Journal on Software Tools for Technology Transfer 6(3), 231–244 (2004)
47. Gönczy, L., Déri, Z., Varró, D.: Model Driven Performability Analysis of Service Configurations with Reliable Messaging. In: Proc. of Model Driven Web Engineering Workshop (MDWE 2008) (2008)
48. Gönczy, L., Déri, Z., Varró, D.a.: Model transformations for performability analysis of service configurations, pp. 153–166 (2009)
49. Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 342–356. Springer, Heidelberg (2009)
50. Ráth, I., Ökrös, A., Varró, D.: Synchronization of abstract and concrete syntax in domain-specific modeling languages. Journal of Software and Systems Modeling (2009) (accepted)
51. Klein, P., Nagl, M., Schürr, A.: IPSEN Tools. In: [59], pp. 215–266. World Scientific, Singapore (1999)
52. Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Wimmer, M.: On Models and Ontologies – A Layered Approach for Model-based Tool Integration. In: Proceedings of the Modellierung 2006, pp. 11–27 (2006)
53. Klar, F., Rose, S., Schürr, A.: A Meta-model Driven Tool Integration Development Process. Lecture Notes in Business Information Processing, vol. 5, pp. 201–212 (2008)
54. Amelunxen, C., Klar, F., Königs, A., Rötschke, T., Schürr, A.: Metamodel-based Tool Integration with MOFLON. In: International Conference on Software Engineering, pp. 807–810. ACM, New York (2008)
55. The TOPCASED Project: TOPCASED - The Open-Source Toolkit for Critical Systems, http://www.topcased.org/
56. Canalsm, A., Le Camus, C., Feau, M., et al.: An Operational Use of ATL: Integration of Model and Meta Model Transformations in the TOPCASED Project. In: Ouwehand, L. (ed.) Proc. of the DASIA 2006 - Data Systems in Aerospace Conference, European Space Agency, p. 40 (2006), http://adsabs.harvard.edu/abs/2006ESASP.630E.40C
57. The ModelWare Project: ModelWare - MODELling solution for softWARE systems, http://www.modelware-ist.org/
58. The MODELPLEX Project: MODELPLEX - Modeling Solution for Complex Systems, http://www.modelplex-ist.org/
59. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook on Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)

The Architecture Description Language MoDeL

Peter Klein

pk@pk-1.de

Abstract. This paper is devoted to the topic of architecture modeling for software systems. The architecture describes the structural composition of a system from components and relationships between these components. Thereby, it provides a basis for the system's realization on technical as well as on organizational level.

We present some key concepts of the architecture description language MoDeL (Modular Design Language). By selecting and combining modeling elements which proved to be helpful for the design of software systems, this approach is integrative and pragmatic: It allows the definition of "clean" logical structures as well as adaptations necessary due to implementation constraints. Both the logical architecture as well as concrete architectures reflecting respective modifications are considered as individual results of architecture modeling. Even more, the transformation steps describing the changes induced by a particular realization constraint contain valuable modeling knowledge as well.

1 Introduction

The observation that the structure of a software system as defined by its architecture is a crucial aspect in the development and maintenance process is almost as old as the software engineering discipline itself. The reasons for this are obvious: The main expenditure during the development process concerning manpower, time, and money is still Programming in the Small (PiS). A good design makes PiS easy in that the programmer can concentrate on a problem with a comprehensible complexity. Errors made in the implementation can be found and eliminated more easily because realization details are encapsulated. For the same reason, software systems are more adaptable, reusable, and portable than before. On the other hand, errors made in the design may lead to an enormous waste of implementation efforts. One may argue that the same dependency holds for requirements specification and architecture, but a good and adaptable architecture always represents a set of similar requirements. To a certain extent, changes in the requirements are readily integrated into a good design.

Specifically with the advent of object-oriented specification approaches, however, there has been an overall tendency to focus more strongly on analysis activities and consider architecture modeling more as a mapping of "classes and objects found in the vocabulary of the object domain" [1] to an implementation view – something which might or might not make sense from a structural perspective. Although languages like UML [2], [3] most certainly can be used to describe software architectures in the above sense, they do not specifically encourage architects to design a robust framework structure for the PiS phase.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 249–273, 2010.

As a language naturally influences the way the speaker thinks when he communicates in that language, it should neither restrict nor overtax the speaker with its vocabulary and rules. For an architecture description language (ADL), this means that it should noticeably …

1. … be easy to use and to understand.
2. … provide the necessary detail to allow the definition of independent working packages for implementation, documentation, and testing, but not more.
3. … not impose a certain style or methodology by preferring or neglecting certain kinds of abstractions.
4. … be independent of the programming language to be used for implementation.
5. … allow for different levels of abstraction.

Based on a long history of preceding work (cf. e.g. [4], [5], [6], [7], [8], [9]), [10] suggests an ADL called MoDeL (Modular Design Language) developed adhering the above requirements. Although retaining the general approach that an architecture, in the first place, needs to define structure, it provides additional views allowing the architect to communicate design decisions. This paper summarizes the main concepts of this language; examples of its use can be found in [10].

2 Architecture Views

One of the basic ideas of the MoDeL language is the distinction of two dimensions with respect to what and how the architecture is modeled, cf. fig. 1. In the top-left corner, a system's static structure is defined with its components and their interfaces. To describe the dynamic behavior of the system, one or more interaction diagrams (top-right corner) may be used. Both specifications are restricted to the logical level, i.e. they strictly adhere to the concepts of modularity and encapsulation. Declarative semantics may be defined formally in the static part and informal operational semantics in the dynamic part as considered appropriate by the architect.

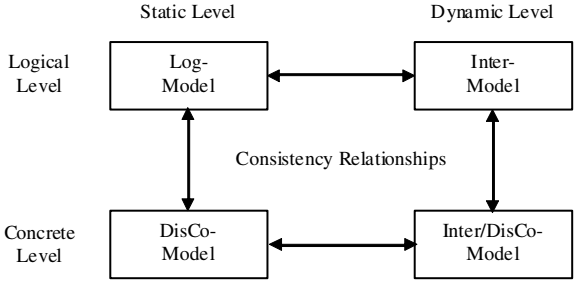


Fig. 1. MoDeL Views

However, there are many potential reasons why an architecture cannot be implemented exactly the way it is specified on logical level. Furthermore, there might be additional information the architect wants to specify beyond the logical structure. Some examples are:

1. Annotation of concurrency properties of components, like which components comprise a process, synchronization semantics of interface operations etc.
2. Introduction of components to handle distribution, e.g. for parameter marshaling, finding a service provider etc.
3. Extension or adaptation of the architecture in order to integrate components with a different architectural structure, e.g. if external libraries or components generated by external tools are used.
4. Specification of the implementation of usability relationships, e.g. via (remote) procedure calls, exceptions, interrupts, event-triggering, or other forms of callback mechanisms.
5. "Opening up" an abstract data type to increase efficiency.

In some cases, there are dependencies between the corresponding realization decisions: As an example, the architect might want to separate a data object (e.g. a database) from the rest of an application to make it remotely accessible by different users. He could implement the incoming usability relationships as RPCs (cf. 4). This also requires corresponding components as those mentioned in 2 to be introduced. These components might use existing data conversion libraries and generated stubs, so he may have to make some changes to the logical architecture as in 3. Consequently, he might want to denote the synchronization semantics necessary due to the concurrent access to the data object as in 1 etc.

All of these activities require changes to the architecture of the system as interfaces change, new components are introduced, or implementation details are added. The resulting architecture though has a different quality than the logical architecture: It does not aim at the best possible structure with respect to maintainability etc., instead it describes a step towards a concrete implementation of the system. In this sense, we call it a concrete architecture in the following.

To some extent, this idea is similar to the distinction between Platform Independent and Platform Specific Models in Model Driven Architecture [11] and its evolutions. However, logical architectures in MoDeL are not necessarily platform independent – this will depend, on a case by case basis, on the nature of the system to be described (cf. section 6).

It should also be noted that different concrete architectures for one logical architecture can exist. These may reflect a sequence of possibly interdependent decisions as sketched above, a set of independent decisions, or different realization variants. Even more, in the context of re- and reverse engineering, the existing system to be analyzed can be considered an implementation of some concrete architecture which has to be derived from the source code or other documentation. Then, the logical architecture can be distilled from the concrete architecture which, in turn, will probably be the basis for restructuring the system and respective new concrete architectures.

Although logical and concrete architectures of a system are naturally related, they should be treated as individual results of the design process. The logical architecture is necessary to understand the system's structure and it remains the central document for implementation and maintenance activities. The aspects described by the concrete architecture are an important step towards the realization of the system, but they should not undermine its logical structure. In other words, each of the architectures represents a set of orthogonal design decisions which should be made, described, and

maintained separately. The integration of these architecture views and checking their consistency can be supported by tools.

And finally, also the transformation steps leading from one architecture to another contain important design knowledge. For many existing systems, only either the initial (and typically logical) architecture is kept or the latest instance of the concrete architecture is maintained. However, both the original design decisions as well as how and why modifications were made are necessary to understand a system's structure, and an explicit transformation step offers a convenient place to document differences between the logical and concrete level. Furthermore, if a specific transformation occurs frequently, it can be possible to formalize it and provide tool support for its application. In this sense, the design knowledge concerning how to modify an architecture to meet some certain purpose can be formally specified and, in consequence, communicated, reused, and possibly supported automatically by tools. Generally formalizing and even automating such transformations, as in [11], has however not been our ambition.

From the wide range of possible uses of concrete architectures, we particularly focus on the specification of concurrency and distribution. These important aspects of a software system require some additional architecture language elements to be described properly; we summarize the respective extensions in the Dis(tributed)Co(ncurrent)MoDeL sublanguage. At the bottom of fig. 1, we have therefore introduced a concrete level which allows such specifications. As on the logical level, static (left box) and dynamic (right box) properties can be defined.

Though fig. 1, at first sight, closely resembles approaches like the 4+1 view model as described in [12], the distinction between the views in MoDeL follows a different pattern: It is not our aim to serve different stakeholder's requirements, but to facilitate communication between the architectural and the "programming" level of a software project. So, for example, the logical view in [12] would remain, in our approach, in the analysis part of the project, whereas the physical view might or might not be described in a concrete MoDeL architecture – depending on whether the distribution of processes across nodes is part of the design or part of the implementation, i.e. whether it is relevant for the overall structure of the system or not.

3 Abstraction Types

As far as its basic abstractions are concerned, MoDeL is mostly motivated by programming language concepts. Apart from being well-known to architects and programmers, this particularly facilitates implementing a design in some programming language. However, all of these concepts have to be reconsidered on architectural level. Of course, implementing a design is more or less problematic depending on the expressiveness of the programming language to be used, but generally always possible. As an example, a module or package concept as can be found in languages like Modula-2, Modula-3, and Ada makes the translation of architectural modules into programming language constructs easier, but a module/package in one of these languages is not necessarily a module on architecture level. Vice versa, an architectural

module can be translated into programming languages without explicit support for such a concept as well.

On component level, MoDeL distinguishes module types in two dimensions:

- Functional and Data Abstraction: Whereas traditional design methods like Structured Design [13] tend to produce hierarchies of functional components in a top-down fashion, object-oriented methods like Object-Oriented Design [1] focus on a loosely coupled set of abstract data types/classes. Meanwhile, it is commonly agreed upon (cf. e.g. [14]) that both kinds of abstractions are necessary. This conclusion is essentially based on Parnas' salient observation that each design decision should be encapsulated by a module and vice versa [15]. Such a design decision can be expressed in MoDeL, according to its nature, either as an abstraction from operation or from state.
- Type and Instance Abstraction: Unlike most other approaches, MoDeL allows modules encapsulating a state or control flow and modules offering a template/type to dynamically create a state/control flow at runtime in one architecture description. Using modules on instance level, the designer can introduce global (i.e. system-wide) state in the architecture without the need to define a data type and additionally assuring that exactly one instance of that type will be instantiated (cf. the Singleton pattern in [16]). Furthermore, this allows a clean model of situations close to the hardware boundaries of the system where concrete devices (e.g. "the keyboard", "the heat sensor") are involved.

Concerning component relationships, the following concepts are supported:

- Locality: Derived from block-structured programming languages as can be found e.g. in the ALGOL family, the concept of locality together with a corresponding set of visibility/usability rules is equally important on architecture level. It denotes that some component is designed to fulfill its purpose in a special context only and that it should not be accessible from outside this context.
- Generality: To introduce components with a general character into an architecture, a corresponding general usability relationship may be used. As an unspecific way to make interface resources of one component usable by another component, this relationship is comparable to an import or use construct in programming languages with a module concept like Modula-2, Modula-3, or Ada.
- Specialization: Object-oriented programming languages like C++, Smalltalk, or Eiffel offer inheritance between data types/classes as a special notion of similarity. Although some of the potential of inheritance depends on using a programming language which supports inclusion polymorphism (in the sense of [17]) in its type system, the concept of modeling some data type as a specialization of another data type is generally useful on architecture level.

Finally, MoDeL supports parametric polymorphism (as defined in [17]) in the form of generic components. Although genericity can be simulated using inheritance on programming language level, these concepts are rather different on architecture level. Whereas specialization is used to model the similarities and differences between data types, genericity allows arbitrary parameterization of arbitrary components.

4 The Static View

MoDeL's key elements can be informally defined as follows:

- **Interface:** A collection of resources like operations, types, constants etc. Basically, we use the term as known from programming languages with a module concept. In contrast to other approaches, we do not assign descriptions of dynamic aspects to operations; consequently, there is no execution model for (a collection of) interfaces.
- **Module:** A module is a logical unit of a software system with a clearly defined purpose in a given context. It consists of an export interface defining which resources the module offers to the rest of the system, an import interface defining which resources from other modules the module may use to realize its export interface, and an implementation in some programming language. The term interface of a module, without further qualification, refers to the export interface.
- **Subsystem:** A collection of components (see below). Subsystems have, like modules, interfaces. The import interface of a subsystem is the union of all import interfaces of the contained components, minus the resources defined by components in the same subsystem. The export interface of a subsystem is an explicitly defined subset of the union of all export interfaces of the contained components.
- **Component:** A module or a subsystem.
- (Component, Module, Subsystem) **Relationship:** A dependency between components resulting from the fact that some resource contained in the export interface of one component (the resource provider) is usable by another component (the resource employer) by appearing in its import interface.
- **Architecture:** The structural building plan for a software system. It defines all of the system's components and their relationships in the form of their import and export interfaces, but not their implementations.

MoDeL distinguishes between different module and relationship types based on the different abstractions introduced in section 3.

4.1 Functional Abstraction

In general, functional abstraction is at hand if a module has some kind of transformation character. This means that an interface resource transforms some kind of input data into corresponding output data. Functional abstraction facilitates the hiding of algorithmic details of this transformation.

An important property of functional modules is that they may not contain memory unless some code inside the module's body is executed. In other words, as soon as the execution of an interface resource of the module is finished, the module has no knowledge about previous calls. The reason for this restriction is that a functional module with a state commonly contains two implementation decisions in one module: one for the actual functionality of the module and one for the state of the module, which should be modeled as a separate data abstraction module instead. This is no conceptual restriction because any module with an internal memory can be made stateless if the state is stored elsewhere and either read by the functional module or passed to respective operations by the resource employer.

This restriction does not mean that the body of a functional module may not have any global variables: it is acceptable if e.g. a module computing trigonometric functions uses an internal table of key values, or if a text-processing module uses a buffer of cached characters. This is no violation of the principle that the mapping of input to output values is independent of the runtime "history" of the module (though it may depend on the history of data abstraction modules used by the functional module). In fact, as the architect defines only interfaces, he cannot keep the implementation from using global variables for whatever purpose as long as the semantic restrictions mentioned above are adhered to.

Regarding the distinction between type and instance level abstraction, we note that the logical level only requires function object modules. Function type modules come into the picture in the context of concurrency, cf. section 6.2.

4.2 Data Abstraction

Data abstraction is present if the module encapsulates the access to some kind of "memory" or "state". The module hides the realization of the data representation. The module's interface only shows how the data can be used, not how it is mapped onto the underlying storage.

In MoDeL, data abstraction is supported by two module types, namely data object and data type modules.

Data object modules represent global state in the architecture which can be implemented by global variables inside the module, i.e. directly mapped onto some structures of the programming language, and/or other data object modules or instances of data type modules. The interface of a data object module exports operations to manipulate the state of the module.

Data type modules export exactly one type identifier (although trivial "helper" types are allowed as well) and access operations for instances of this type. Other modules may use the interface to create instances of the data type and manipulate it with the given operations. Data type modules are templates for the creation of memory and may not contain a global state: The execution of an interface resource on an identical instance of the type always modifies this instance in the same way. As with function object modules, this does not mean that the body of the module may not contain global information, but operations on one instance of the type may not have visible side-effects on other instances.

For reasons of clarity, we always treat types introduced by abstract data type modules as reference types. This does not necessarily require such a type to be mapped onto a "pointer" in the programming language: A CORBA object identifier, an integer-valued handle etc. have reference semantics as well. Important consequences of this restriction are:

- Instances of the type always have to be created (and possibly destroyed) explicitly, be it on the heap provided by the programming language runtime system or within some other component of the software system. Noticeably, assignment (explicit or by parameter passing) or variable declaration never create a new instance of the type. Means to copy instances, if required, have to be specified as an operation. This allows the architect to specify whether copying is possible at all and, if so,

what semantics (deep, shallow, or anything intermediate) is used. Multiple copy operations with different semantics can be specified as well.
- Comparison is reference comparison. As above, the specification may define if comparison of values is possible and what semantics apply by defining respective operations.
- Using reference semantics makes it sensible to think of `self`, the parameter denoting the object on which the operation is applied, as an input parameter: Whether or not the operation modifies the instance's state, `self` as a reference is never changed by the operation.

4.3 Interface Extensions and Export Control

Frequently, the necessity arises to give some employers of a module more visibility on its realization than others. A common example is that an employer defining a subtype of some data type requires more control over the attributes of its supertype than "ordinary" employers of that module. Exactly this situation is covered in many object-oriented modeling and programming languages by introducing a separation between public, protected, and private resources of a data type.

Although the public/private/protected scheme meets many basic requirements, it has some major drawbacks:

- The separation is only usable in the context of data types and inheritance. There is no good reason why a similar mechanism should not be available for other module types or other module relationships as well. In object-oriented literature, this is typically solved by using the "friend" concept. However, when using friend relationships, export control is given up altogether as complete access to the realization details is granted.
- Even under the above restriction, there are situations that cannot be described appropriately with only three layers of accessibility. This results from the approach to define the accessibility of a feature as an inherent property of the feature itself: In the interface of the module, the feature is generally classified as being public, protected, or private – regardless of the employer. Conceptually, this is not quite correct as its accessibility has something to do with how the feature is used in the overall design of the system, i.e. the actual pair of provider and employer.

Consequently, we take on a more general approach that allows one or more views on a module to be specified. Some views might provide more control over the module's internals than others by relieving certain abstractions. To provide such views, we use the concept of interface extensions: the architect may define an arbitrary number of additional so-called view interfaces for the module. A view interface may contain additional operations and helper types for data object and functional modules. Furthermore, for data object modules, some or all of the internal representation of the module's state may be exposed. For data type modules, additional features (including attributes) may be defined.

4.4 Module Relationships

After presenting the different kinds of basic design units, we now introduce the means MoDeL provides to describe interactions between these units. First of all, some modules

want to make use of resources offered by other modules. We distinguish three different logical levels on which such dependencies can be discussed:

1. The first prerequisite for the interaction between an employer and a provider module is that the design allows the employer to access the resources offered by the provider. We call this the usability level.
2. If the architecture allows some module to access another module (on the usability level), the employer may make use of this by actually using some of the provider's resources (e.g. if the employer's implementation contains a call of a procedure offered by the provider). This use is static, i.e. it can be determined by looking at the implementation of the employer. Therefore, it is on the static uses level.
3. If a static use of some provider resource is executed during the runtime of the program, we say that this is on the dynamic uses level.

Since we are considering the static view on the architecture, it cannot be determined whether the employer's implementation makes use of a resource or not. It can only allow some module to use another module's resources. So, if we talk about some module importing another module, this is always on usability level. The uses level, however, plays a role in a dynamic view of the system, cf. section 5.

Besides the usability relationships from above, we also have structural relationships between modules. These are used to express structural design concepts and allow or forbid certain usability relationships.

4.4.1 Local Containment/Usability

We start our discussion with a relationship called local containment. This is a structural relationship describing that some module is contained in another module. From this containment, some rules derived from the block-structuring idea present in many programming languages follow. This relationship forms a local containment tree in the module dependency graph. Placing a module in such a tree means that the module can only be accessed from certain parts inside this tree. In this sense, it introduces information hiding on architecture level.

The following usability relationships are possible in a local containment tree: A module can use itself, its direct successors, its predecessors, and direct successors of all predecessors (especially its brothers). This is completely analogous to the rules of locality and visibility in block-structured programming languages. We say that potential local usability exists between the module and its above mentioned relatives in the containment tree.

One problem with the relationship of potential local usability is that many relationships are made possible between modules which are not necessary. We therefore introduce the local usability relationship. Local usability is a relationship which is explicitly specified in the architecture, although such a relationship may only be defined between modules for which a potential local usability exists. In other words, the local usability relationships are a subset of the potential local usabilities defined by the designer.

4.4.2 General Usability

The local usability relationship introduced so far is not suited for all situations where one module wants to use resources from another module. This is particularly the case

if some module should be usable by arbitrary employers, possibly from different containment structures.

With the general usability relationship, the architect has means to express that a module exports general resources which can be accessed from all other modules for which a corresponding general usability relationship exists.

There is no structural relationship directly connected to general usability: General usability edges from any part of the system can end in one module (which, itself, must not be contained in another module). On the other hand, it is not unusual that only one employer uses some provider module through general usability. Whether or not a module is generally or locally usable is a question of the kind of provider module. If a module offers general services to employers, it should be inserted using the general usability. If it offers services which are only useful in a certain context, it should be inserted using the local containment and local usability relationships.

4.4.3 Specialization/Specialization Usability

The other structural relationship in MoDeL besides local containment is the specialization relationship. Although the concept of specialization is influenced by ideas from object-oriented programming languages, it denotes a relationship between modules on the design level here. Whether the actual implementation language's type system supports specialization in some way should not influence the logical architecture (though it might influence a concrete architecture).

The specialization relationship can only exist between data type (and function type, cf. section 6.2) modules. If some data type is a specialization of another data type, this implies that every instance of the special type has at least all the properties of an instance of the general type. As usual, we call the special type a subtype of the general type, which is vice versa referred to as the supertype of the special type. This terminology extends to arbitrary ancestors and predecessors of a data type module in the specialization hierarchy.

An important characteristic of the specialization relationship is that the set of features offered by the subtype is a superset of the set of features of the supertype. We therefore do not have to repeat these features in the subtype's interface.

As with local containment and local usability, the structural specialization relationship is accompanied by a usability relationship, the specialization usability. By definition, a specialized module needs to import the module it specializes.

It should be noted that the structural relationship of generalization implies no usability relationships. In MoDeL, a subtype module may not use some supertype's operation just because it is a subtype: All usability relationships have to be introduced explicitly by the designer. On the other hand, the structural relationship again determines the set of possible usability relationships. This leads to a strong correlation between architectural usability relationships and the common notion of an "import": A subtype module needs to import the supertype module's interface in order to use the corresponding type identifier in the declaration of the subtype identifier. Furthermore, if some subtype wants to call an implementation of an operation as defined for some specific supertype (in contrast to whatever implementation is dynamically bound to the operation, see below), it has to address this module directly. Obviously,

every subtype module must have a specialization usability edge to its immediate supertype module and may have additional specialization usability edges to arbitrary predecessors in the specialization hierarchy.

Apart from these definitions on module level, we also have to consider the connection between specialization as a module relationship and the consequences for the types and their operations exported by corresponding modules. Unfortunately, this affects issues which are related to Programming in the Small. On one hand, defining them on architectural level makes a few (possibly unwelcome) assumptions concerning the implementation. On the other hand, in the context of specialization, a data type module's interface cannot be properly defined and understood if these questions remain open.

- As usual in object-oriented modeling approaches, we consider variables of an abstract data type as polymorphic, i.e. any variable or formal parameter of some type can be assigned/passed a value of some subtype of this type.
- A supertype operation's implementation can be redefined for a subtype. Syntactically, this is not visible in the interface, but an informal (comment) or formal (cf. section 4.7) semantics specification can indicate this.
- In the context of polymorphism, a variable's or parameter's dynamic type at runtime can be a subtype of its static type (according to the declaration). If the subtype has redefined operations, we always consider the dynamic type decisive for the selection of the implementation (late binding, dynamic dispatching). In this case, there may be transfers of program control from one module to another module without an explicit usability relationship between them: An employer module can call a data type's operation according to the (supertype) interface it is aware of, whereas at runtime, some code within another provider module (defining a subtype of the type in question) is executed. Conceptually, however, the employer is independent of the subtype interface and only depends on the supertype interface. Therefore, a usability relationship between employer and subtype module is not required.
- We allow operations to be redefined in subtypes, i.e. a subtype may supply a new (new in the sense of a different signature and different semantics) operation with the same name as an inherited operation. It should be noted that redefining an operation is something different than redefining an operation's implementation as discussed above. Although operation redefinition is not unproblematic with respect to polymorphism, it is frequently the case that some subtype operation is "more complex" than a very similar operation in a supertype, consequently, it will want to raise more exceptions, might need more input parameters, produce more complex results etc. Some of these points can be relieved by stating co- and contravariance rules for parameters, but this introduces additional complexity without completely solving the problem.

It should be noted that redefining an operation, in contrast to redefining the implementation, does not replace the existing operation's implementation. As far as the selection of the implementation in the context of polymorphic variables is concerned, we consider the static type of the variable as decisive: This is necessary as an employer can only provide the input and handle the output of an operation as it is specified in the interface it knows and by which it declares the variable/parameter.

Frequently, if a subtype redefines an operation, it also redefines the implementation of the inherited operation of the same name to sensibly handle attempts of operation calls which are not appropriate for the dynamic type of an instance.

4.5 Subsystems

Obviously, the module level introduced so far is too fine-grained for the description of large software systems. We therefore need design units which allow a hierarchical specification of the architecture. Subsystems allow the designer to express such units which are "greater" than modules: they may contain an arbitrary number of modules and other subsystems.

Most of the characterizations given for modules at the beginning of this chapter can be applied to subsystems as well: first of all, subsystems are units of abstraction. They have an interface which describes the resources which can be accessed from the outside. The interface of a subsystem is a composition of explicitly selected modules and/or subsystems inside the subsystem. Furthermore, subsystems are units of work, units of testing, and units of reusability. For this reason, subsystems should – just like modules – obey the rules of low coupling and high cohesion: the modules and subsystems should not interact more than necessary with other units on the same design level. On the other hand, a module or subsystem should only contain logically related resources.

However, it still makes sense to distinguish between modules and subsystems as the former are directly linked to the PiS level in the sense that they involve implementation work, whereas the latter start out purely as a design concept which might not materialize in program code at all.

As stated above, the designer decides explicitly which interfaces in the subsystem contribute to the subsystem's interface. An immediate consequence is that we cannot assign each subsystem a unique type as we could with modules. We therefore do not distinguish different subsystem types. Nevertheless, we sometimes talk about functional or data type subsystems if the subsystem's interface consists of one or more modules of the corresponding type.

Of course, being part of a subsystem is a structural relationship between the corresponding components. Accordingly, we already introduced a natural candidate for subsystems in the previous chapter, namely local containment trees. Containment trees are a special sort of subsystems where the interface is implicitly given by the root of the tree. However, apart from isolating components within the subsystem from components outside, containment trees additionally introduce the locality/visibility restrictions as described above.

4.6 Generics

MoDeL supports a reuse mechanism commonly known as genericity (or parametric polymorphism according to the more precise terminology introduced in [17]). The main idea of genericity is to write (generic) templates for system components. In the template, an arbitrary number of details is not wired into the code, instead the template code refers to these details using formal parameter names. The programmer can then create a concrete component by supplying the missing details in the template.

We call these details (generic) parameters, and the process of creating a component using a generic template and generic parameters (generic) instantiation. Accordingly, the resulting component is called a (generic) instance.

Ideally, the process described above is directly or indirectly supported by the programming language or the development environment. But even if not, it allows the architect to indicate that for different physical modules the same code should be used.

A common example for generic templates is a collection (container) of instances of an entry type. The generic parameter for the template is the entry type for the collection. The reason for this approach is that the implementation of the collection is more or less independent of the type of objects it can store, i.e. neither interface nor implementation of the collection refer to special properties of the entry type. MoDeL therefore allows a generic specification of the container, making the entry type a generic parameter. The designer can then use this template to instantiate arbitrary concrete containers by providing respective entry types from data type modules in the architecture.

We can readily extend the concept of generics from modules to subsystems. Conceptually, generic subsystems are just as important as generic modules, e.g. in a situation where some collection is built on top of one or more other collections. For example, a generic module offering a table data structure might be implemented using a generic data type module for dynamic arrays. In this case, it would be natural to instantiate a generic subsystem consisting of a generic table and a generic array module.

Of course, a generic subsystem may also contain non-generic components. From the viewpoint of the instantiation process, they can be thought of as generic components without generic parameters. Furthermore, in order to make sense, a generic parameter of the subsystem must be generic parameter for at least one generic component within the subsystem. Vice versa, a generic parameter of a contained component must be a generic parameter of the subsystem unless the component is already instantiated within the subsystem.

4.7 Specifying Semantics

Up to now, the MoDeL language contains constructs to define components, their interfaces, and their relationships. Such descriptions, however, are worthless if the architect has no way to attach some “meaning” to the respective specification elements. In general, there are two levels on which the architect will have to consider such semantic issues:

- On the level of single interface definitions, the architect will have certain semantics in mind when specifying operations. Depending on the context, he will add e.g. a corresponding informal comment to the operation describing the desired behavior.
- On the level of components and their interaction, the architect influences the possible implementations by the static structure he uses. Considering the example of implementing some table module on top of a module for dynamic arrays, a suggestion for the implementation of the table has already been provided. To some extent, this can be considered as a violation of the abstraction boundary between Programming in the Large and Programming in the Small, but it is inherent to the notion of software architectures as a building plan [18].

Consequently, the architect should have means to specify what he had in mind when defining an architecture in a certain way. We can distinguish two dimensions for such additional information:

- Black-Box vs. White-Box View: The black-box view defines the semantics of interface operations in terms of interfaces only. It makes no assumptions concerning the implementation of an operation and if or how an operation could be realized (in general using interfaces of other components). A white-box view, sometimes called glass-box view, defines the behavior of operations in the context of the overall system, i.e. it does include interactions between components. In short, the black-box view is concerned with what an operation/component does and the white-box view with how it is done.
- Formal vs. Informal/Semiformal Notations: In practice, it is common to use informal plain text comments in natural language to describe the semantics of interface operations. This is probably sufficient in many situations, but certainly lacks the preciseness necessary in others. The most important reason for taking a more formal approach is to avoid misunderstandings between architect and programmer, but it may serve other purposes as well: A formal specification can be subjected to formal consistency and completeness checks, it can be used to simulate/prototype the specified system (parts), to formally prove the correctness of hand-coded implementations against the specification, or to generate test cases for such an implementation.

It should be noted that, according to whether a black-box or white-box view is taken, the term "semantics" as used above takes on two different meanings: In a black-box view, it generally denotes a mapping of input to output values of each operation. Accordingly, we call this the declarative semantics of a component. In a white-box view, on the other hand, the architect might specify a specific sequence of steps that needs to be taken inside the implementation. This kind of definition is generally called the operational semantics of a component/operation. Both declarative and operational semantics can be defined formally or informally.

Another property which can be assigned to the definition of a component's semantics is whether it is complete or not. By complete we mean that the semantics of the component (as the sum of all of its operations) is specified for all possible input values and states. Noticeably, the definition of the component's semantics may cover only "some" (important, frequent, critical etc.) cases. In practice, interface comments mention only the intended "usual" behavior. The behavior in other cases is undefined. Typically, informal specifications tend to leave missing details to the intuition of the reader.

In the following, we will mainly use the term semantics as referring to declarative semantics. Since the declarative semantics of a component and its operations are static properties, they will be discussed next. Operational semantics are concerned with the runtime behavior of the system; they will therefore be a topic in section 5. It should be noted that this distinction is not always clear; the terms semantics and behavior are indeed closely related. Consequently, other authors use different definitions and classifications in this context. For our purpose, it is sufficient to bundle the terms static specification (or, to be more precise, specification of static properties) and declarative

semantics on one hand and behavior specification (specification of behavioral properties) and operational semantics on the other. Typically, operational specifications in our sense include information about the interplay of components whereas declarative specifications are "context-free", i.e. do not refer to the specification of (operations of) other components. Another characteristic property of operational specifications is that, if the specification is sufficiently complete and formalized, it can be used to prototype or simulate the system.

As far as corresponding specification languages are concerned, it is frequently not a question of language constructs but of how the language is used in order to determine whether they specify what is done or how it is done. However, formal or semi-formal languages for operational specifications often have some execution semantics which, itself, may be defined formally or semi-formally. Taking the term "specification" in a broad sense, a high-level imperative programming language can be considered an operational specification language where the language and its execution semantics are mostly defined semi-formally.

Informal semantics specifications are widely used, e.g. in the form of comments. These specifications take, in general, a black-box view on the component, are more declarative than operational, and are almost always incomplete in the above sense. For obvious reasons, they do not need further discussion here.

More interesting is the question of what approach should be taken if a formal specification is required. Since we focus on architecture modeling, it is neither possible nor sensible to discuss this question in the general context of formal system specifications. Analyzing comparative case studies (cf. e.g. [19]) shows that it is hardly possible to judge the expressiveness of formalisms without a specific intention in mind. To clarify the key requirements for a formalism suitable for our purpose, we can state the following:

1. The formalism should be able to express the semantics of (elements of) MoDeL component interfaces. Noticeably, there should be a clear mapping between interface resources and specification elements.
2. Specifications should be static by nature, we are not (yet, cf. section 5) considering control flow issues, component interaction, or algorithms.
3. We are not (yet, cf. section 6.1) considering a possibly concurrent and/or distributed realization of the system. Terms like processes/tasks, synchronization, timeout etc. play no role in our logical architecture view and should therefore be avoided in a corresponding specification.
4. It should be possible, but not a requirement to formally specify the complete system.

We can inductively conclude:

- As given by 1, we mainly want to describe the semantics of operations in a component's interface. From 2 follows that we can describe an operation only in terms of states. The state space should provide means to define the applicability of an operation (preconditions), an operation's impact (if any) on the current state (postconditions), and an operation's result. These states must be abstract in the sense that they are meaningful with respect to the interface. Since the interface, in turn, consists of operations, nothing should be in the state space which is not necessary for the specification of operations in the above sense.

- Considering MoDeL's module types as introduced above, it is obvious for data object and data type modules that the abstract states which can be used in a formal specification are equivalence classes of physical states of corresponding data objects and data type instances. For specification purposes, a data abstraction module (being specified as abstract states and operations on these states) is frequently viewed as an abstract machine. For functional components, a more difficult situation arises since there is no associated state. There is no general solution to this problem: If, for example, a functional component's operation computes the square root of a positive real number, it is probably not necessary at all to provide a formal specification: An informal appeal to mathematical intuition will sufficiently explain the input/output relation of this operation. If the component is used to make complex transformations on a data object/data type instance or to transform one data object/data type instance into another, it could be desirable to specify its semantics in terms of the related input/output data structures. Finally, if the component's purpose is to set up a certain control flow among subtasks, static specifications are by definition not appropriate.

Especially points 2 and 3 from the above list show that the intention behind many existing formal approaches like CSP, Esterel, FOCUS, Estelle, LOTOS, or SDL is not quite in line with our intended purpose. For the remaining formalisms, we have to answer the question of how states are represented. We consider two options:

- In algebraic approaches ([20]), states are expressed implicitly, i.e. the state change induced by some operation is defined by how it affects subsequent operations.
- Model-based approaches like Z, VDM-SL, or B/AMN explicitly define the state space, usually in terms of typed set theory.

Basically, both approaches fulfill the requirements from the above list. However, in practice, model-based approaches tend to be easier to understand: Defining states explicitly allows the reader to focus on one operation at a time instead of having to understand an operation in the context of the other operations in an interface.

Eventually, among the model-based formalisms, we have chosen a systematic use of Z [21] to accompany MoDeL interface definitions. The main reason for this decision was that all other formalisms with a broader acceptance include imperative constructs to cover operational aspects of semantics (so-called wide-spectrum formalisms). For our purposes, this is not required and might even lead to "overspecified" interfaces.

5 The Dynamic View

As was pointed out in the previous section, it is necessary to document the semantics of a software system on two levels: Declarative semantics to understand the purpose of a component and its interface resources and operational semantics to understand the relationships between components and their interplay. Formal and informal declarative semantics definitions can be used in MoDeL in the form of Z specifications and comments, respectively, augmenting interface specifications with the corresponding information. Here, we take up the discussion of specifying operational semantics.

A variety of specification languages for the semantics of software systems and their components have been developed over the past years. One major direction originates in the structured approach to requirements analysis and design with its data, functional, and control model of a system ([22]). From the respective languages ER (and its many extensions), SA, and SA/RT, the latter two can be allocated to behavioral descriptions. The second direction emerged under the collective term object-oriented modeling and produced a wide variety of approaches [23] adopting and redefining structured concepts as well as adding new notations. In particular, we have considered the Object Modeling Technique (OMT) [24], the Object-Oriented Analysis/Design (OOA/D) method proposed by [1], and the Unified Modeling Language UML [2], [3].

Based on this work, we have defined an approach similar to Collaboration Diagrams as in OOA/D for the MoDeL sublanguage InterMoDeL for describing operational semantics of architectures. InterMoDeL specifications are called interaction diagrams. Like CDs, interaction diagrams allow the description of exemplary behavior. However, InterMoDeL is based on interacting components, not on interacting objects. This noticeably means that instances of data type modules are not shown as nodes in interaction diagrams (although they may appear as parameters in operation calls).

Interaction diagrams are composed of the following elements:

- Nodes in the diagram are components (modules or subsystems) as defined for architecture diagrams.
- Components may be connected with uses relationships. Uses relationships indicate the call of an operation offered by the resource provider. Obviously, for a direct uses relationship to be legal, a corresponding usability relationship in the architecture diagram must exist. Every uses relationship features one or more labels which, in turn, consist of a sequence number and an operation name with a signature.

Every label reflects a call of the operation as indicated by the label. The “actual parameters” shown with the call must match the formal parameter signature as given in the textual interface specification of the called component. Sequence numbers in InterMoDeL interaction diagrams are hierarchical as in UML.

It should be noted that, by assuming reference semantics for data type modules, there can be no “hidden” calls to a module. As an example, complex data types frequently require a used-defined implementation to copy an instance of this type. Using value semantics, this operation may be called implicitly by an assignment operator or by passing a variable of that type by value in a procedure call. In contrast, with reference semantics, every call inducing the execution of code within the data type module refers to an operation with a name and signature explicitly defined in the features part of the interface. This enhances the readability of corresponding interaction diagrams and makes consistency checks between architecture diagrams, interaction diagrams, and module implementations easier.

Interaction diagrams may also contain indirect uses relationships. They feature the same kind of label as direct uses relationships have. An indirect uses relationship indicates an operation call where the resource employer is not aware of the component in which the operation is implemented. This occurs e.g. when some component calls an operation in another component via a callback (procedure pointer) which

itself has been passed to the calling component previously. For an indirect uses relationship, there need not be a parallel usability relationship. In fact, it frequently happens that indirect uses appear in the reverse direction to a "corresponding" usability relationship.

Another (technically closely related) example for indirect uses relationships appears in the context of specialization/polymorphism. When a component calls an operation of a data type, the implementation actually executed by this operation call might be located in a specialized data type module. According to the intended semantics of the diagram, the architect may use either a direct uses relationship to the operation of the supertype the calling component statically imports or an indirect uses relationship to the subtype operation which is actually called.

6 Concrete Architectures

As was already mentioned, there is a number of reasons which make it reasonable to modify a logical architecture described with LogMoDeL and InterMoDeL as a first step towards an implementation. We have introduced the term concrete architecture to reflect the results of such steps. The specific transformations from logical to concrete architecture are driven by specific needs, and they may or may not be applicable independently. As a consequence, the concrete architecture will evolve in a sequence of incremental transformation steps from the logical architecture. All of these steps select a realization variant for some abstract situation given by the logical architecture. In this sense, a logical architecture contains the logical essence of a host of concrete architectures describing special realization choices.

Introducing logical and concrete architectures allows us to model a software system on a pure structural level without having to implement it exactly that way. On the other hand, adaptations or extensions of the architecture necessary to fulfill certain realization constraints are still planned and prepared during the design process (and not, as frequently found in practice, as an "on-the-fly" activity during the implementation phase). Furthermore, logical and concrete architectures as well as the dependencies and differences between them are described and documented separately: Both architectures as well as their correspondences represent individual design decisions made by the architect.

Still open, though, is the question of which architectural concepts are on logical level and which are on concrete level. There is no general answer to this question; what is considered "logical" or "concrete" strongly depends on the respective software system and its properties. A rough guideline results from distinguishing between "inherent" and "derived" properties of the system:

- We can characterize a property of a software system as "inherent" if it is tightly connected to the problem to be solved. For example, a business administration system will always store its data in some sort of database, a CAD tool will always have a graphical user interface, a screenshot capturing tool will refer to some particular window system, and a telecommunications switching system will always be a concurrent system. In general, it is not very useful to leave corresponding design decisions to the concrete level as this overloads the logical level with abstractions of no practical use.

- "Derived" properties are not introduced by the problem itself but by a particular solution. Taking up the examples from above, we can state that the choice of a concrete database system and an appropriate database schema are derived design decisions for the architecture of a business administration system. The same holds for the selection of a particular GUI for a CAD tool or a certain window system version for the screenshot capturing tool. Corresponding architecture modifications should be made on the concrete level.

So, for example, being a concurrent system is probably an inherent property of e.g. a telecommunications switching system while it would be a derived property of e.g. a compiler. Accordingly, respective specifications will occur on logical level for the former and on concrete level for the latter type of software.

6.1 Concurrency

Although, in this sense, architecture language concepts for concurrency and distribution are not generally allocated to logical or concrete level, we discuss them in the context of concrete architectures. As our approach origins more from the area of main-stream application systems than from embedded or real-time systems, concurrency and distribution are typical derived properties: They are introduced to increase performance or to allow different users access to shared information, but they are not part of the problem domain. The respective modeling concepts to be introduced in the following, however, can be used on logical level as well.

If we now want to describe concurrency on architecture level, we first have to decide what a concurrent component actually is. The notion of active objects is frequently used in current literature to introduce concurrency on design level. Since the term active generally appears in different semantics, we sketch some definitions here.

1. [1] defines active objects as objects which encompass an own thread of control. This definition and others closely relate threads or processes with single components, noticeably data type or data object modules (classes/objects). A similar approach, although not restricted to data abstraction components, can be found in the task concept of the Ada programming.
2. Functional modules are abstractions of operations; they have no state and serve transformation or controlling purposes. Data abstraction modules encapsulate state or a template for state; they hide the internal representation of this state and allow access to it only via a set of corresponding operations preserving the abstraction's semantics. In this sense, functional components act and data abstraction components are acted upon, and it seems natural to apply the term active to functional and passive to data abstraction components.
3. When two components interact, e.g. by a procedure call from one component into the other, the called component performs some action on behalf of the caller. Like under point 2, the calling component acts and the called component is acted upon, so we may apply the terms active and passive accordingly. We can readily generalize this scenario to all situations where one component induces some activity in another, be it directly (e.g. procedure call) or indirectly (e.g. event triggering).

4. A special variant of 3 concerns the situation when two components interact for execution control purposes, i.e. one component manipulating the thread of control of another component (start, terminate, suspend, resume etc.). Semantically, this is on a different level as 3, although it may be realized in just the same way (e.g. by procedure calls). Therefore, we distinguish the corresponding notions of active and passive in the thread controlling sense from those concerning the actual abstraction of a component.

Obviously, 1 directly introduces concurrency on design level. 2 and 3 describe definitions which can be applied to sequential systems as well. Point 4 is a rather complicated case because the passive component in this sense has to have a thread for the definition to make sense; therefore it is an active component in the sense of 1. The active component in 4 might or might not be active in the sense of 1. Be this as it may, the definition in 4 already requires some notion of concurrency. To avoid confusion with other semantics of active and passive, we use the term acting component for a component which has an own thread of control and reacting component for a component which has not.

Choosing the semantics described in 1 as the foundation for our terminology does not mean that the other semantics for active and passive play no role in the design of an architecture description language for concurrent systems. In fact, relating the notions of acting and reacting components to active and passive components in the sense of 2 to 4 yields some valuable ideas presented in the following.

Comparing the definitions of acting and reacting to those of functional and data abstraction components shows that there is an intuitive relationship between the property of owning a thread of execution and the abstraction decision represented by a module's type. If a data abstraction component's purpose is to hide the details of some state's internal representation, to define what operations are semantically sensible on the state, and to map these operations onto some manipulation of the internal representation, we observe that there is no need for such a component to own a thread of control. The execution of access operations of the component is always (possibly indirectly) triggered by some functional component and can therefore be performed in the corresponding thread of control. On the other hand, the execution of transformation or controlling activities may or may not happen concurrently. Therefore, we restrict acting components to functional abstractions, i.e. functional components may or may not be acting, while data abstraction components are always reacting.

Relating our definitions to the active/passive semantics described by 3 leads to the question whether an acting component can be acted upon in the sense of (directly or indirectly) calling one of its interface operations. A negative answer would result in the restriction that an acting component may have no interface. In the reverse direction, we see that a component with an empty interface surely has to be acting: If it has no own thread of control and no other control flow can enter through the interface, the component is obviously useless. But may an acting component have an interface and be passive in the sense of 3? The intuitive notion that a component either does something of its own or it acts on behalf of other components leads to the conclusion that this is not the case.

Considering some typical examples of functional abstractions, we observe that there are indeed some sensible interface operations for acting components. But we also note that these are operations which directly or indirectly manipulate the component's

thread of control. We therefore allow exactly such thread controlling operations in the interface of acting components, even though the control semantics of an interface operation might be implicit.

6.2 Function Type Modules

We can now come back to the question of function type components. It may seem trivial at first sight that it is possible for an operation in an acting component's interface to create a new thread of control instead of manipulating a thread encapsulated by a function object. Up to now, we have considered acting components on instance level, i.e. equipped with exactly one thread of control. But just like we do for data abstraction components, we can shift the concepts for functional components to type level. This allows us to distinguish between function object and function type components: The former represent a single (set of) computation(s), and the latter are templates for instantiating such computations at runtime. In sequential systems, we do not need to bother with function types because a single instance of every function is sufficient. However, in a concurrent system, function types are necessary since their instantiation is the logical counterpart of the creation of a new thread of control.

Considering the similarities between data and function type modules, the question arises how far the analogy between these component types goes. More precisely, on architectural level, we have to define whether function type instances are allowed as parameters in operation signatures and whether there is a useful meaning for specialization relationships between function type modules.

We can readily resolve this issue with a simple construction: Data type module instances carry a state and operations manipulating this state. Function objects may have no state, so function type instances have no state as well. However, the thread of control attached to a function type instance represents a state with respect to the underlying execution engine, e.g. an execution point, a call stack etc. This thread state can be interpreted as the counterpart of a data type instance state.

6.3 Synchronization

To consider the implications of having several acting components in a software system, we first have to define more clearly what is meant by a thread of control and a process. Without being precise about this term so far, we have implied some independence concerning their execution. However, taking a closer look, different degrees of independence can be identified.

According to the extent of context information carried along with a thread of control, we distinguish three levels of processes:

- A thread of control consists of an execution point ("program counter"), a call stack, and an address space. In this case, the terms task, heavy-weight process, or simply process are frequently used. For a heavy-weight process, all state information, be it local (in a procedure) or global, can be accessed only using the control flow of that process.
- A thread of control consists of an execution point and a call stack. This is generally denoted as a light-weight process or thread. Local state information, as it is stored in the call stack, is uniquely instantiated and associated with every thread. Global state, however, is accessible by all threads, possibly at once.

- A thread of control exists of an execution point only. All state information is shared amongst all "processes". This case is rather rare in mainstream application systems, though it may occur in embedded real-time systems.

In the following, we use the term process as referring to any of these forms.

Whatever approach is taken, at certain points in time, some processes will generally have to cooperate in some way. Cooperation involves one or more of the following aspects:

1. Passing of information between processes may occur with a call of an interface operation. The details of what information is exchanged (input/output/exception parameters with their respective types) are covered by the signature of the operation and therefore already part of the architecture specification of an interface as introduced so far. For light-weight processes, arbitrary data may be communicated. Heavy-weight processes may not exchange references to data inside their address spaces, i.e. only values of atomic types or references to function type instances can be passed. Several techniques exist to handle the communication of more complex data structures by "flattening" the information into a stream of atomic values (marshalling) on employer side and recomposing the complex value on provider side (unmarshalling).
2. Synchronization of access to shared data, i.e. which operations of a component may be called concurrently and which may not, is not apparent from the interface so far. Furthermore, it has to be denoted whether the employer or the provider is responsible for ensuring that the synchronization rules are followed. We will extend interface specifications to express this information below. Note that this is not relevant for heavy-weight processes. For light-weight processes, the corresponding information must be given for data abstraction components. In the context of fly-weight processes, functional component specifications might be subject to synchronization extensions as well.
3. For control flow organization, we make the assumption that some process interaction in which the provider process is not able to accept a request will suspend the employer process until the request can be handled. For heavy-weight processes, this may occur if the thread of control in the provider process is currently executing some other code. For light-weight processes, the same may happen if the provider component has the responsibility of synchronizing access to its state and another control flow currently holds access to resources which may not be used concurrently with the requested resources. In both cases, we assume that the request is queued, the requesting process suspended, and the queued requests are handled in the respective order by the provider process, thereby resuming the control flows of the requesting processes.

To specify synchronization constraints as required by point 2 above, we have introduced a virtual resource model akin to the concept of monitors [25]. This model provides means to abstract from the state of a data object or data type instance in the form of so-called mutexes (this terminology is derived from the threading facility of the Modula-3 programming language). Synchronization requirements can be expressed as dependencies of operations from these mutexes.

A mutex can be declared in an interface. It represents a resource or set of resources which can be accessed by at most one process at a time. An operation may depend on a resource on two levels: If the execution of an operation demands that the resources represented by a mutex should be made available to one process only, we interpret this as the operation "locking" the mutex. Every mutex can be locked by at most one process. To be more precise, only one process at a time may execute an operation which locks a certain mutex. Other processes needing access to the same mutex have to wait for the locking process to end the execution of the respective operation. Frequently, however, a process does not have to lock a mutex, it is sufficient for the operation's execution that it is not locked by any other process. In this case, we say that the operation "requires" the lock. Locking a mutex is a stronger condition than requiring it, i.e. an operation locking a mutex requires it as well. Finally, a process cannot lock a mutex as long as an operation is executed which requires it. MoDeL furthermore allows the architect to specify whether the provider module or the employer component(s) is responsible for protecting the mutex.

A noticeable advantage of the explicit modeling of virtual resources with mutexes, apart from the possibility to specify fine-grained access control, is the ability to express inter-module synchronization schemes. Mutexes, as anything else in a component's interface, can be imported by other interfaces and may be used to formulate synchronization requirements there.

7 Summary

This paper summarizes some of the key concepts of the architecture description language MoDeL as introduced in [10]. Its basic properties are:

- The understanding of the contents and purpose of a software architecture is that of a blueprint for the system under consideration. Its core abstraction results from the distinction between an interface and its realization.
- The basic unit of modeling is a component that may range in size and complexity between a module and a complete software system. The abstraction provided by a component depends on its logical characteristics and purpose in the context of the system.
- The language itself is multiparadigmatic. It supports different kinds of abstractions for components and relationships, modeling on different levels of the logical hierarchy, and multiple levels of detail between pure logical structure and actual realization structure.
- Different views concerning static and dynamic as well as logical and physical properties are supported. The relationships between (elements of) these views are precisely defined.

References

1. Booch, G.: Object Oriented Analysis and Design with Applications. Benjamin/Cummings, Redwood City (1994)
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1999)

3. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading (1999)
4. Altmann, W.: A New Module Concept for the Design of Reliable Software. In: Raulefs, P. (ed.) Workshop on Reliable Software, pp. 155–166. Hanser-Verlag, Munich (1979)
5. Gall, R.: Structured Development of Modular Software Systems – The Module Graph as Central Data Structure. In: Proceedings of the Workshop on Graphtheoretic Concepts in Computer Science 1981, pp. 327–338. Hanser-Verlag, Munich (1982)
6. Lewerentz, C., Nagl, M.: Incremental Programming in the Large: Syntax-Aided Specification Editing, Integration, and Maintenance. In: Shriver, B. (ed.) Proceedings of the 18th Hawaii International Conference on System Sciences, Honolulu, vol. II, pp. 638–649 (1985)
7. Lewerentz, C.: Extended Programming in the Large within a Software Development Environment. ACM SIGSOFT Software Engineering Notes 13(5), 173–182 (1988)
8. Nagl, M.: Softwaretechnik: Methodisches Programmieren im Großen. Springer, Berlin (1990)
9. Börstler, J.: Programmieren-im-Großen: Sprachen, Werkzeuge, Wiederverwendung. Umeå University Report UMINF 94.10, Doctoral Dissertation, Aachen University of Technology, Umeå University (1994)
10. Klein, P.: Architecture Modeling of Distributed and Concurrent Software Systems. Doctoral Dissertation, Aachener Beiträge zur Informatik, Band 31, Wissenschaftsverlag Mainz in Aachen, Aachen (2001)
11. Kleppe, A.: MDA Explained, The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
12. Kruchten, P.: Architectural Blueprints—The "4+1" View Model of Software Architecture. IEEE Software 12(6), 42–50 (1995)
13. Stevens, W., Myers, G., Constantine, L.: Structured Design. IBM Systems Journal 13(2), 115–139 (1974)
14. Perry, D., Wolf, A.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
15. Parnas, D.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), 1053–1058 (1972)
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, Reading (1995)
17. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys 17(3), 471–522 (1985)
18. Kiczales, G.: Towards a New Model of Abstraction in Software Engineering. In: Yonezawa, A., Smith, B. (eds.) Proceedings of the International Workshop on New Models for Software Architecture 1992; Reflection and Meta-Level Architecture, Tokyo, pp. 1–11 (1992)
19. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
20. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 2 – Module Specifications and Constraints. Springer, Berlin (1990)
21. Spivey, J.: The Z Notation – A Reference Manual, 2nd edn. Prentice Hall, New York (1992)
22. Kohring, C.: Ausführung von Anforderungsdefinitionen zum Rapid Prototyping – Requirements Engineering und Simulation (RESI). Doctoral Dissertation, Aachen University of Technology. Shaker-Verlag, Aachen (1996)

23. Hutt, A. (ed.): Object Analysis and Design – Description of Methods. Wiley, New York (1994)
24. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs (1991)
25. Hoare, C.: Monitors: An Operating System Structuring Concept. Communications of the ACM 17(10), 549–557 (1974)

Towards Managing Software Architectures with Ontologies

Marcel Bennicke and Claus Lewerentz

Brandenburg University of Technology,
Konrad-Wachsmann-Allee 1, 03046 Cottbus
{mab,cl}@informatik.tu-cottbus.de

Abstract. Software architectures are key enabling assets within organizations that develop complex software systems. Among other purposes, software architectures are useful to maintain intellectual control over a software product. We propose a method to continuously check the consistency between a specified architecture model and structural information reverse engineered from the code. We develop criteria that a design language for architectures should fulfill and show that an ontology based description has substantial benefits over the standard modeling languages MOF/UML/OCL. Using ontologies allows the explicit modelling of architectural styles as well as concrete system structures in a single architecture design language. The resulting specifications are modular, compositional and evolvable. Using ontologies we can apply an ontology reasoner to implement consistency checks. Our method integrates previously separate checks such as checking for allowed dependencies and coding style into a single framework and enables more powerful and flexible analyses.

1 Introduction

Software architecture centric development processes have been proposed to overcome the tremendous gap between requirements and system implementation [3]. A software architecture is the first available consistent set of design decisions that describe a solution for the stated and implied requirements on a yet abstract level [3]. Among many other purposes, it serves as a reference throughout the entire system lifecycle. It defines the fundamental, high-impact structures of a system, the implementation platform and prescribes patterns and rules that allow developers to implement the functionality using the platform.

This intended use of an architecture can only be successful, if the specified architecture is consistent in itself and is eventually followed throughout the entire development process. Brooks pointed out that *conceptual integrity* is the most important goal in system design [9]. In brief, conceptual integrity means that recurring types of problems are solved in a similar and consistent way. This applies to both, the end-user perceivable properties of a system (e.g. the way system functions are presented to the user), as well as to the internal system structures (e.g. the way the system is structured and how cross-cutting services are integrated).

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 274–308, 2010.

1.1 Architecture Centric Development Processes

In order to motivate our approach and to illustrate its relationships to other development approaches this section provides a brief overview of architecture centric software development.

Process Phases and Important Artifacts of Software Architecture. Figure 1 shows the role of software architecture in the context of an architecture-centric development process and the constituent parts of a software architecture. The system analysis phase identifies and records the set of requirements a system should fulfill, identifies the major business objects and captures them in a domain model. This initial activity is also to discover possible design constraints[1]. The architecting phase creates a software architecture description. It specifies the fundamental abstractions, the system structures and an 'implementation plan' according to the specific requirements and known design constraints. The implementation phase eventually creates the executable software system and through tests it delivers evidence that the initially stated requirements have been met.

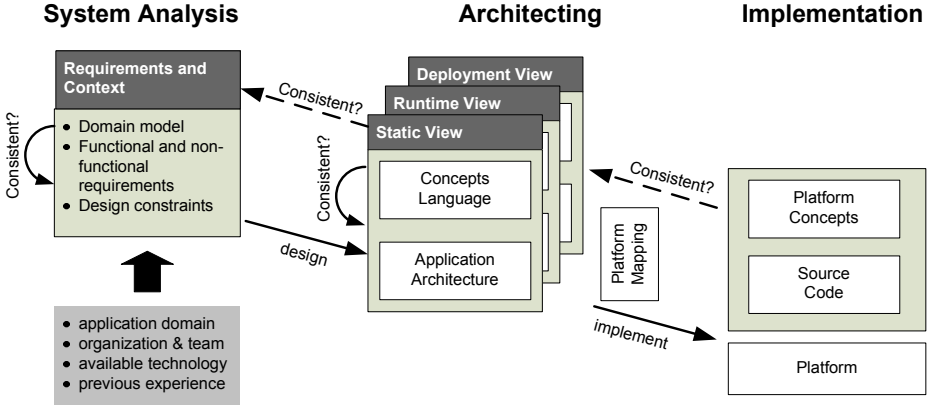


Fig. 1. Process elements of an architecture-centric development process

In literature there is consensus that software architectures are best described using multiple views [15, 3, 18]. Typical such views comprise a static view showing the static structures of a system, and a runtime view showing the processes and objects existing at runtime (see architecture views in figure 1).

The static view is probably the most common view for developers and architects as it decomposes the system in a way that is relevant for their work. For example, such a view shows modules, interfaces and relationships between them. Developing the static view has two aspects. First, the functionality stated as

[1] Design decisions with a pre-determined outcome.

requirements has to be subdivided into coherent chunks. In the *application architecture*, the architect specifies which concrete components the system consists of and assigns responsibilities and relationships to them.

A second, sometimes implicit decision regards the language which is used to express the application architecture. There are two major options: one may use a general-purpose language such as the UML [10] or develop a domain specific concepts language that suits the particular requirements [33]. Concepts are types of things from which a system is constructed. Architectural concepts are technically motivated. Examples of concepts are component, layer, process, message queue, filter or persistent entity. We and others [8, 23, 33] argue that clarifying the concepts is an integral part of developing a software architecture. Concepts are specific for a system or a family thereof, because they result out of requirements which are specific for a system or a system family. Dealing with them consciously and capturing them in a (formal) language supports the processes that need to deal with concepts (such as system design, knowledge transfer, code generation, code quality assurance). We also argue that concepts and a concepts language are not only a useful utility to design new systems, but they can also be reconstructed for existing systems. Concepts are naturally used by software engineers and programmers to design solutions to recurring design problems. However, very often concept properties are not clearly defined or communicated or get lost over time such that the code only roughly reflects the design ideas of the original concepts. For example, according to our experience [5], many systems have a claimed layered architecture, but the actual dependency structure found in the code only roughly reflects this idea.

The architecture concepts language is a domain specific language (DSL) with limited generality which provides the architect just with the concepts needed to describe the system's application architecture. It also contributes to the establishment of a clearly defined design terminology that transports design ideas within the development team. When a new system is created, an architect should ponder about useful concepts that can help to realize the requirements. Architectural styles, patterns, previous designs are sources of concepts to be included in the language. Sometimes they also arise bottom up, when for a system currently under development, a recurring solution pattern emerges. Despite some concepts appear to be generally applicable (such as interface), it is important to reconsider and explicitly state the structural and behavioral semantics of every concept for the current context. For example, regarding interfaces the architect may have to think about whether a call-by-reference or call-by-value semantics is needed, whether a component offers one or several interfaces, whether an interface allows for synchronous or asynchronous communication, how error situations are communicated consistently and much more.

Various authors underline that the essence of an architecture should be decided upon and described without assuming a particular platform [23, 28, 32]. This makes the architecture transferable to new platforms an makes the design decisions embodied in the architecture explicit, comprehensible and traceable.

Because this platform independent description (encompassing a platform independent application architecture and a platform independent conceptual architecture) eventually has to be implemented using a concrete platform, a *platform mapping* is needed. The architect needs to decide about

1. a suitable platform and
2. a strategy to implement the so-far abstract architecture using the platform

The mapping specifies how the platform-independent elements translate into platform-specific elements and which services of the platform should be used to implement the application architecture. OMG's Model Driven Architecture vision [23] also follows such a distinction between platform-independent models (PIM) and platform-specific models (PSM).

The three artifacts concepts language, application architecture and platform mapping are commonly considered to be architectural artifacts [32] (cf. figure 3).

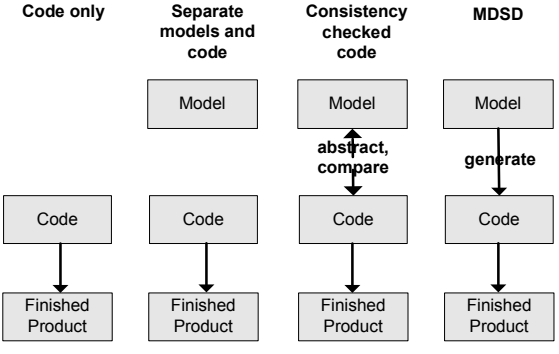


Fig. 2. Relationships between architectural models, code and software products in different development approaches

Implementing an Architecture Centric Development Process. Opposed to e.g. agile development processes, architecture centric development processes accept that in order to efficiently build high-quality software products, a model[2] of the future product is needed.

Figure 2 shows how the relationship between architecture models, code and the finished product can be set up in different development approaches. In our view, each of these options has benefits and bears some risks depending on the particular situation. The 'separate models and code' approach is very flexible but has the drawback that the produced code tends to drift away from the model over time disqualifying the model as a useful documentation of the system. Model driven development enables great efficiency improvements but can only play to this strength, if a stable concepts language can be identified early such that

[2] Abstract representation of an existing or envisioned object or process using either non-formal or formal language.

the cost of creating the necessary language infrastructure (editors, generators) breaks even for the number of systems built with it. Also, the models need to be made sufficiently detailed to allow for comprehensive code generation.

In this text we focus on the 'consistency checked code' approach that creates and maintains architecture models and program code in separate processes and aligns them only through analytical measures. The observation that code tends to drift away from a once specified architecture is a well known phenomenon. The reasons are manifold and range from lack of communication, quick fixes built under time pressure and lack of time for re-documenting a deliberate deviation. However, allowing temporary inconsistencies also has its benefits, because it allows solutions to unexpected design problems to be explored before a rather long-term architecture or code generator change is complete. Especially in new domains it is difficult to design an appropriate concepts language in a 'big bang' approach as required by generative model driven development.

In such contexts it is important to clean up inconsistencies regularly because otherwise long-term goals such as maintainability, conceptual integrity, comprehensibility, reusability and system evolvability are at stake. Automated consistency analyses support such clean-up activities. Consistency analyses can also be combined with generative model driven development, because the still necessary hand-written code can potentially drift away from the intended architecture as easily as in the 'separate models and code' approach. Eventually consistency analyses can help in restructuring a legacy system where control over the system structures has been lost.

1.2 Requirements for an Architecture Design Language to Support the Consistency Checked Code Approach

Independent of the development approach chosen (cf. figure 2), a software architecture needs to be developed and specified. Requirements for the specification documents itself depend on the organizational and development process activities that shall be supported by them. In the following, we derive requirements for an architecture design language from these groups of impact factors. Figure 3 shows that some requirements for such a language apply generally – to any approach depicted in figure 2. Others are specific for the consistency checked approach we like to address.

Analyses to be supported. In a consistency checked code approach we understand a software architecture model as a template which guides other activities but which is not used to create other artifacts directly. Generally, the models should serve the tool supported analysis for several types of inconsistencies. This alone entails the need for a formal, machine-processable architecture design language. Besides that, formal modeling tremendously increases the accuracy of specifications and, therefore, contributes to the quality of specifications.

Figure 1 indicates that in architecture centric development processes, several types of feedback loops are important for product quality and development

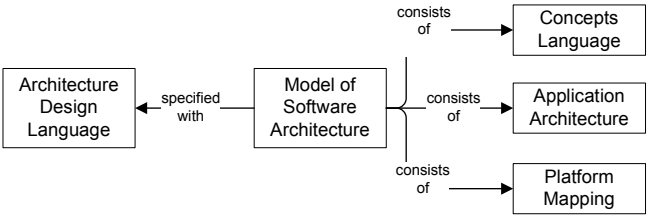


Fig. 3. Overview of terms used in section 1.2

efficiency. In every process phase, the created results should be evaluated for correctness and consistency[3]. In the architecture development phase the concepts language, the application architecture and a platform mapping are specified. Each of these parts specifies a set of constraints that the eventually implemented system should fulfill. Especially for complex systems and in the presence of reuse and composition of specification modules, there are good chances that constraints make contradictory statements. Hence, the architecture design language should support the architecture development phase by enabling the following two types of analyses.

- *Concepts consistency*: Analyze the concepts language for inconsistencies.
- *Application architecture consistency*: Analyze the application architecture for inconsistencies with respect to the concepts language.

As an example for concepts consistency suppose a concepts language with the terms repository, business component, report component, a use relationship and the following set of constraints:

1. a repository is only used by business components
2. a report component uses at least one repository
3. something cannot be both a business component and a report component

This constraint set is inconsistent, because the first two constraints force an asserted report component to use a repository, which in turn may only be used by a business component. This is only satisfiable, when the assumed component is both a business component and a report component. On the other hand, constraint 3 states just the opposite.

As an example of an inconsistent application architecture suppose the above concepts language and the following application architecture specification

$$\mathsf{report\ component}(\mathit{reportgen}),\ \mathsf{repository}(\mathit{data}),\ \mathsf{uses}(\mathit{reportgen}, \mathit{data})$$

This specification is inconsistent because is forces a report component to use a repository which is illegal according to constraint 1.

The second type of inconsistency indicated by figure 1 regards inconsistencies between artifacts created in different development phases. In the consistency

[3] Being free of contradictory statements.

checked code approach the architecture design language must particularly enable the third type of analysis

– *Architecture-code consistency* Analyze the source code for inconsistencies with respect to the application architecture and the concepts language.

Architecture-code consistency can only be checked with the help of a platform mapping and a code model. The code model represents the structural information found in source code, i.e. in Java classes, packages, method calls or inheritance relationships. The platform mapping describes how source code elements and their relationships relate to their architectural counterparts. To continue the above example assume a mapping to the Java platform where the repository(*data*) is implemented by a Java class(*DataRepository*) and report component(*reportgen*) is by the classes of a Java package(*reportgen*). Furthermore the platform mapping translates every code-level dependency (e.g. method call, class inheritance relationships, access to attributes) into the architectural use relationship. We call the result of applying the (reverse) platform mapping to an extracted code model, the extracted application architecture. Then, the following extracted code model would be considered inconsistent with the concepts language, because it translates into the same (erroneous) extracted application architecture as above.

$$\mathsf{package}(\mathit{reportgen}),\ \mathsf{class}(\mathit{DataRepository}),\ \mathsf{call}(\mathit{reportgen}, \mathit{DataRepository})$$

Eventually a code model is inconsistent with a specified application architecture, when the extracted application architecture misses some of the elements or relationships specified in the application architecture.

Requirements originating from system modeling and analysis. The syntax and semantics of language constructs are always driven by the domain that shall be described using the language. In the case of a software architecture design language the domain consists of all conceivable concepts languages, application architectures and platform mappings.

To capture all three aspects a very expressive design language or set of languages is needed. In particular, the design language must allow for the specification of a language (the concepts language) and sentences in that language (the application architecture) at the same time. For example, we want to be able to describe that a system has a component concept and at the same time that it has the concrete components *a* and *b*. Furthermore, the comprehensive specification of architectural constraints must be supported. Example types of constraints are participation of concepts in relationship types (the provides relationship type connects a component and an interface), cardinality restrictions (a component provides at least one interface) and restrictions on primitive data types such as strings (a class implementing an entity must follow the regular expression `'.*Entity'`).

Our consistency checked code approach shall be much more lightweight than a generative model driven approach. We want to support situations where the concepts language part of the specification can be changed by the architect

easily. This would make the method applicable to reverse and reengineering situations where the architect starts with an initial hypothetical architecture and adjusts it as he learns about the system. In generative model driven development the concepts language is well supported by a language infrastructure (editors, generators), which however cannot be changed easily. Depending on the concrete technology, the cycle of changing the concepts language is long. Favre has also noticed that when a system evolves, the languages used to describe it evolve too [12]. We need a *lightweight language infrastructure* (editor, constraint checker) that supports short evolution cycles.

Organizations that build many different software products should try to standardize and reuse the architectures of their systems in order to increase their efficiency. The design language should allow *specification packages* to be created independently and to be composed into new specification packages. Every specification package is a collection of constraints. For example, a specification package A could define the properties of component based architectures. Such a specification package would introduce the concepts component and interface. Another specification package B could define the properties of layered architectures. A third package C could reuse the previous both packages A and B to define a concrete application architecture that is both a component based and a layered architecture.

Furthermore we require *support for abstraction and refinement*. When building a concepts language it is helpful to be able to express hierarchies of concepts. A refined concept has to fulfill all the constraints defined for the related abstract concept but may have additional constraints associated. For example a specification package to describe 'component based' architectures introduces the concept of a general purpose component with an interface and an implementation part (cf. figure 4). In a specification package for 'distributed information systems' the business component concept inherits all properties of the general purpose component but has additional constraints. Having available such a refinement hierarchy of specification packages opens up the chance for package-based consistency checks. The result of a consistency check would not only be 'yes' or 'no', but 'no, it is not consistent with the online shopping reference architecture, but consistent with the distributed systems architecture'.

As a final requirement the consistency checker must be able to apply *open and closed world semantics* for analyzing the correctness of models. Suppose a concepts language that incorporates an Entity concept and a Repository concept (cf. top part of figure 5). Entities are business objects persisted in a database. Repositories implement basic database operations for entities (create, read, update, delete). An architectural constraint is that every entity must be managed by one repository that implements the operations (cf. cardinalities of the association).

The bottom left part of figure 5 shows a concrete application architecture consisting of only one Entity *a*. Applying standard UML closed world semantics, this would be an invalid model, because the Entity *a* needs a reference to exactly one Repository which is missing. However, as a specification this application

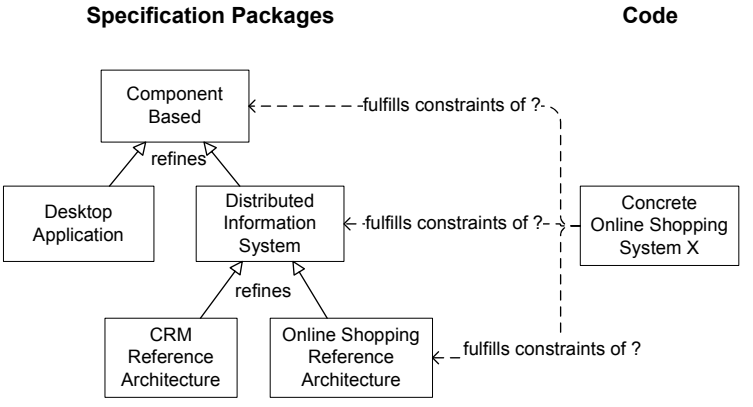


Fig. 4. Example refinement hierarchy of software architecture specification packages

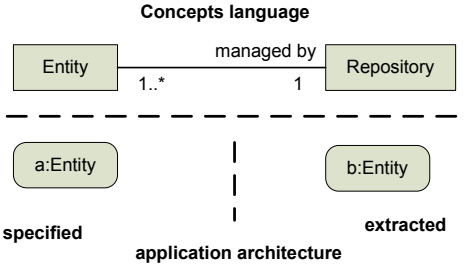


Fig. 5. Example concepts language and two models expressed in the language

architecture model makes sense and should be accepted. From the architect's perspective it is sensible to specify all the concrete entities the system should have, but leave the definition of repositories to developers (repositories are not 'defined' by programmers in the model, but directly created and implemented in the code). The reason for this distinction is that entities represent information which is essential for the system's overall functionality, which is not the case for repositories. Hence, the application architecture model must be evaluated using so-called open-world semantics which flags an error in a model only when the model contains contradictory statements but not when some statements are missing.

The bottom right part of figure 5 shows the same model. Consider this model as the result of extracting the actually implemented architecture from the code (the extracted application architecture). There, we have an Entity with no associated Repository which means that Entity objects cannot be persisted. In this situation the missing association should indeed be interpreted as an error. Applying so-called closed-world semantics to this analysis would yield the desired alert. We will come back to the difference between open- and closed-world semantics in section 3.

Section Summary. To summarize this section, we require an architectural design language with the following characteristics. The following requirements apply to all development approaches.

1. The specification must be modular.
2. The design language must be sufficiently expressive. It must enable the specification of a concepts language, an application architecture and a platform mapping.
3. The following analyses are to be supported: language consistency, application architecture consistency.
4. The analyses shall consider the composition of all specification packages relevant for one system.
5. There must be support to factor out common abstractions and to reuse them in new contexts.

The following items are specific for the consistency checked code approach we like to address in particular.

6. The language must be formal and machine processable.
7. The analysis for code-architecture consistency must be supported (i.e. there must be support for 'executing' a platform mapping).
8. We would like to express constraints about the existence of elements (e.g. there must be a component(*reportgen*)) and about element properties (e.g. a repository is only used by business components).
9. The co-evolution of the concepts language and the application architecture shall be supported by a lightweight language infrastructure.
10. There must be support for different semantics during analysis (closed versus open world semantics).

2 An Example System

The purpose of this section is to introduce the software architecture of a small but sufficiently complex system that will be used to illustrate our approach. This section presents the example using mostly natural language and ad-hoc graphical notations.

The example system is an information system that supports the business processes of a library. The system should enable library staff to manage library users (create/update/remove account for users), enable staff to maintain a media catalog (enter new items, handling of lost or damaged books), provide comprehensive search capabilities with various criteria for library users, and support the process of loaning media (make reservations, borrow a copy, return a copy, create reminders for overdue copies).

The system is to be implemented as a distributed system with local client applications accessing a central server.

2.1 Concepts Language and Application Architecture

As part of architectural design, two models have to be developed simultaneously: the concepts language in order to make the types of things the system is composed from explicit as well as the concrete application architecture in terms of the concepts language.

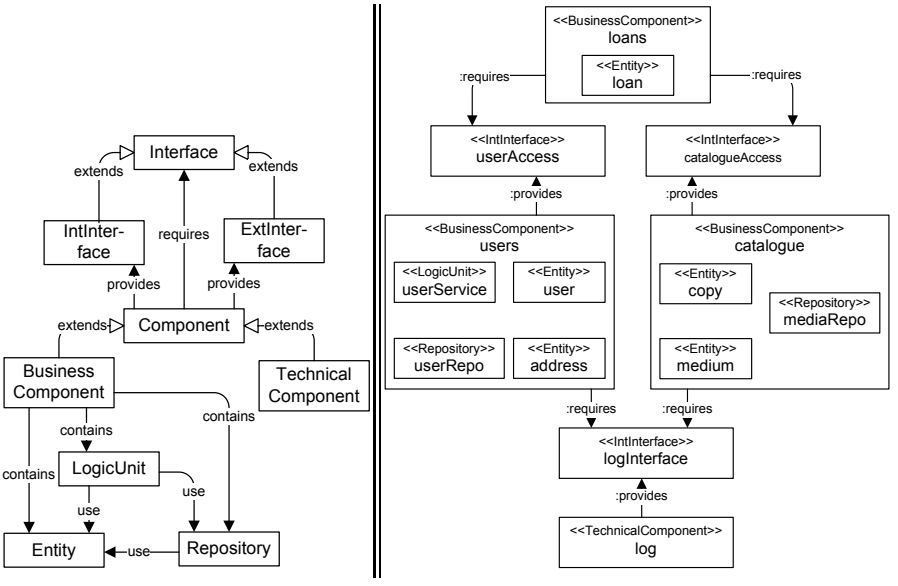


Fig. 6. Concepts language and application architecture for the library system

Figure 6 (left) shows the basic concepts and relationship types of the concepts language for the server-side part of the library system. The semantics of the concepts is as follows: The library system shall be structured as a component-based system that consists of a flat structure of Components. We distinguish application domain specific BusinessComponents and general-purpose TechnicalComponents. A component provides at most two but at least one type of interfaces: an ExtInterface (external interface) with call-by-value semantics and an IntInterface (internal interface) with call-by-reference semantics. A component may also require some Interfaces of arbitrary kind to access the services of other components. It is assumed that the components run under control of an application container that adds container-managed transaction behaviour to the component interface methods. Each BusinessComponents contains at least one LogicUnit. A LogicUnit implements the core business logic. The logic is implemented with the help of Entities (persistable business data structures) and Repositories (modules that implement database create, read, update operations for some entities). A structural constraint is that Repositories and Entities must both reside inside the same BusinessComponent (not expressed in the figure).

Figure 6 (right) shows a part of the concrete application architecture of the library system. Some relevant Entities are *user*, *address*, *medium*, *copy*, and *loan*. The system consists of the BusinessComponents *users* (user management), *catalogue* (functions to maintain the catalogue of media) and *loans* (functions related to loans) as well as a TechnicalComponents *email* (for sending emails). The figure also shows the planned relationships between components (e.g. *loans* may access the IntInterface of *users* but not vice versa).

It should be noted that the application architecture is deliberately incomplete, because the architect specifies only those parts which are architecturally relevant. Only the sets of Components and Entities are completely specified, because defining Components is important to divide the overall functionality into coherent subsets to which programmers can add detail. The definition of Entities is important because they represent the business data of the application which is a precondition for a detailed specification of functionality, e.g. as use cases. On the other hand, the Repositories are not completely specified because they are rather technically necessary and can be neglected for discussing functionality. Nevertheless, they represent an important architectural concept that is necessary to implement data persistency.

2.2 Platform Mapping

The definition of a platform mapping goes hand in hand with the decision for a particular platform. A platform may be used 'as is', but very often it is useful to extend it with constructs that implement the basic behavior of the architecture concepts. For example, an `AbstractEntity` class may provide common services required for all persistent data entities. The generation of unique identifiers, and operations for cloning and equality tests are examples of such behavior. For the Component concept a late binding of required Interfaces is indispensible to enable component exchangeability and for component testing. A late binding would allow a Component to be bound to mock implementations of other Components without changing the production code.

Such extensions to an existing platform as the Java language and its standard libraries influence the platform mapping because certain conventions need to be followed for using the APIs. Second, the demand for a human comprehensible source code also contributes to the mapping. Ideally, architectural concepts are not lost in the code perspective but are visible to the reader. Choosing speaking source code identifiers or using a standardized organization scheme through directories, libraries, packages are options to make architectural concepts visible in the code.

In the library example, the concept of an Entity maps to a Java class

- inheriting from `AbstractEntity` class – the platform extension for entities
- with a name ending with `Entity`
- being contained in the package of the BusinessComponent with which the architecture specifies a contains relationship, i.e. if the BusinessComponent's package is `library.business.x` (due to a BusinessComponent mapping rule), the Entity must be `library.business.x.common`.

Figure 7 (left) lists more example mapping rules for some of the previously defined architectural concepts. The right hand side of the figure displays a fragment of the physical source code layout for the library system that results from applying the platform mapping to the application architecture given in section 2.1.

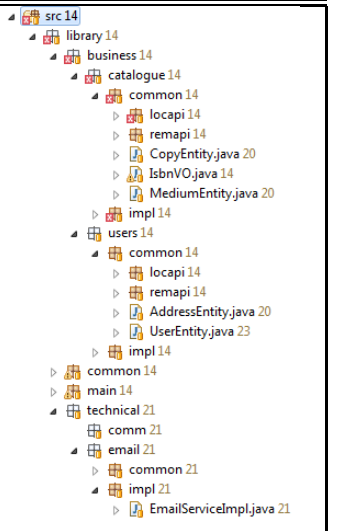
Concept, Relationship	Mapping rules	Example structure
component	a Java package below `library.X.Y` where X indicates the module type (business or technical) and Y the component name. All Java types within this package belong to the component.	
internal component interface	all Java types which are located in the subpackage `common.locapi` of the components's package	
external component interface	all Java types which are located in the subpackage `common.remapi` of the components's package	
component implementation	all Java types which are located in the subpackage `impl` of the components's package	
entity	a Java class whose name matches the pattern `.*Entity` and which is located in the subpackage `common` of the component's package	
use	any form of cross references in the code in the direction from using a program element (method, class, package) to the location of its declaration.	
provides	the Java containment relationship between the containing Component's package and the Interface package.	

Fig. 7. Some mapping rules for concepts and relationship types of the library system's concepts

3 Ontologies as an Architecture Design Language

The previous section has outlined the requirements for an 'architecture design language' and the operations necessary for the models expressed with this language.

In the domain of software engineering the OMG set of languages is the standard approach to modeling. In fact, concepts can be expressed with a meta-model implemented as an UML profile or MOF extension. An application architecture then becomes an instance model that instantiates the elements of the meta-model. Rules can be implemented as OCL constraints. Relationships between different models may be expressed with the help of a model transformation language such as QVT [24] or ATL [16].

3.1 Benefits of Ontology-Based Modeling

However, we propose to use ontology and rule languages originally developed for the Semantic Web [6] because they have three major advantages with respect to the consistency checked code approach we like to address.

1. Ontology languages and ontology tools allow modelers to easily express meta models and models in a single formalism and within one tool context, which is not the case for the OMG languages. There exist comprehensive ontology development suites such as Protégé [31] and TopBraid Composer [30]. Both of these properties enable the lightweight language infrastructure we need.
2. Ontology languages are based on formal description logics which are a subset of first-order predicate logic with the tremendous advantage of being decidable. All ontology interpreters (reasoners) support the *concepts language consistency* check out of the box. They even offer explanations for the constraints that cause an inconsistency. Reasoners also work with an open world assumption by default. Closed world behavior can be enforced through specific description logic constructs (cf. section 1.2 why both kinds of semantics are needed).
3. Ontologies can be specified as modules and there is a standard `import`-statement which fulfills our modularity and compositionality requirements. The module concept allows expressing the architecture specification packages as ontology modules. One can choose to put schema and instance information into a single or in several modules. The same applies to abstract and more concrete concepts. This allows the modeller to freely subdivide an overall specification into modules as needed in order to make them comprehensible and reusable.

The use of ontology technology in software engineering is increasing [11] but far from commonplace and particularly there is no ontology-based approach known to the authors in the realm of architecture-code-consistency checking.

3.2 Ontologies and Rule Systems

Ontologies and rule systems are logic-based semantic web technologies that enable the modeling of knowledge [1]. A principal goal of both is to enable reasoning about the stated knowledge, i.e. to allow specific correctness checks to be performed and to infer new facts from the given ones. A major application area

of ontologies is to describe terminologies of an application domain and to support correctness checks for such terminologies. For example the GALEN project maintains an ontology describing a huge medical terminology [25].

The development of ontology languages and rule systems is driven by the need for an expressive modeling language that is at the same time logically decidable. Unfortunately, decidability limits the structure of facts that can be expressed with either an ontology or a rule system. On the other hand, there are properties that can only be modeled with ontology constructs or only with a rule system. Therefore, both knowledge modeling approaches have been integrated to increase the overall expressivity and enable integrated inference. Expressing the architectural constraints we like to address is also only possible using rules and ontologies. Hence, this section presents a brief overview of both.

Ontologies. Informally, an ontology describes a domain of discourse [1]. It consists of a set of terms and relationship types between terms. The terms represent sets of objects called *concepts*. The objects are called *individuals*. In a business domain concepts are Product, Customer, Employee or Invoice. The relationship types denote semantically meaningful associations between concepts. Example relationship types are pays (a Customer pays an Invoice) and name (a Customer has a name denoted by a `String` data type). Besides defining such terms, an ontology more importantly describes constraints. Different kinds of constraints are supported such as

- subset relationships: a Book is a Product (written $\mathsf{Book} \sqsubseteq \mathsf{Product}$)
- constraints about participation in relationships: an Invoice can only be paid by a Customer (expressed as: $\mathsf{Invoice} \sqsubseteq \forall\ \mathsf{pays}^{-1}.\mathsf{Customer}$).
- disjunction statements: the sets of individuals of Customer and of Employee are disjoint (expressed as: $\mathsf{Customer} \sqcap \mathsf{Employee} \sqsubseteq \bot$)
- value restrictions: the name or a Customer must not be empty (expressed as: $\mathsf{Customer} \sqsubseteq \forall\ \mathsf{name}.$`String[pattern "[a-z]+"]`).

More formally, an ontology $\mathcal{O}$ is a set of so-called axioms and assertions over disjoint sets of concept names N_C, role names N_R and individual names N_I. An interpretation $\mathcal{I}$ is a tuple $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of a non-empty domain $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$. The interpretation function assigns to each concept name $C \in N_C$ a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, to each role name $r \in N_R$ a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and to each individual name $i \in N_I$ an element $i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.

Axioms are constraints that characterize the concepts and relationship types (the so called TBox or the schema part) and assertions denote concrete individuals and relations (the ABox or instance part) of a knowledge base. The most important types of axioms are general concept inclusions (GCIs) of the form

$$C \sqsubseteq D$$

A GCI informally reads as a logical implication ('anything that is a C is also a D'). The C and D can be names for concepts (elements of N_C), but also more complex so-called concept descriptions. For example, the concept description

$$\geq 2\ \mathsf{provides.Interface}$$

denotes all individuals that have two or more provides relationships with distinct individuals of the concept Interface. Assertions state the existence or non-existence of concrete individuals and relations. For example $\mathsf{Component}(x)$ expresses that x is a Component and $\mathsf{provides}(x, y)$ states a concrete tuple of the provides relationship type.

The formal background behind ontologies are description logics (DLs) [2]. The main purpose of a description logic is to enable inferences (reasoning) according to the semantics of the axioms and assertions. For example, if we have a DL program (an ontology) with the statements

$$\mathsf{Component} \sqsubseteq \geq 2\ \mathsf{provides.Interface}, \quad \mathsf{Component}(x)$$

then we can infer the following facts

$$\mathsf{Interface}(y1), \mathsf{Interface}(y2), \mathsf{provides}(x, y1), \mathsf{provides}(x, y2)$$

A specific description logic allows for specific types of axioms to be used. There is a general tradeoff between the expressivity description logics and the complexity of the reasoning tasks performed in such a logic. The description logic community has developed a convention to briefly denote classes of expressivity (see table 1). For example, a $\mathcal{SHIN}$ DL does not allow an axiom of the form Component $\sqsubseteq \geq$ 2 provides.Interface but only Component $\sqsubseteq \geq$ 2 provides, i.e. one cannot express cardinality restrictions and participating concepts at the same time. For other forms of axioms and assertions occurring in different description logics, please refer to literature (e.g. [2]). For our examples, we use the $\mathcal{SHROIQ}(\mathbf{D})$ description logic [13] that underlies the most recent Web Ontology Language (OWL) 2.0 [19]. For brevity we use the description logic notation instead of one of the OWL syntaxes.

For practical applications of ontologies it is important to discuss their *satisfiability*. A given interpretation $\mathcal{I}$ satisfies a concept description C (written $\mathcal{I} \models C$), iff C has a non-empty extension under $\mathcal{I}$, i.e. $C^{\mathcal{I}} \neq \emptyset$. An interpretation $\mathcal{I}$ satisfies a GCI $C \sqsubseteq D$, iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Eventually an interpretation $\mathcal{I}$ satisfies an ontology $\mathcal{O}$ (written $\mathcal{I} \models \mathcal{O}$), iff $\mathcal{I}$ satisfies all axioms from $\mathcal{O}$.

This notion of satisfiability lays the ground for practical ontology reasoners such as Pellet [29] and HermiT [21] to provide a number of inference services which are held to be key for most applications of ontologies.

- *Consistency Checking* verifies the consistency of an ABox with respect to a TBox, i.e. the operation ascertains whether both contain contradictory statements.
- *Satisfiability Checking* is the operation to check whether a concept or all concepts occurring in an ontology have a non-empty satisfying interpretation.
- *Classification*, which calculates the subset relationships $\sqsubseteq$ between all named concepts of an ontology.
- *Realization* computes the $instance - of$ relationship with the most specific concept for every individual. Realization can only be done after classification because the concept hierarchy must be known.

Table 1. Syntax and semantics of various description logic constructs (from [14])

Construct	Syntax		Semantics	DLs
	Expression	Axiom		
universe concept	$\top$		$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$	
empty concept	$\bot$		$\bot^{\mathcal{I}} = \emptyset$	
atomic concept	A		$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	
atomic role	r		$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	
transitive role		$Trans(r)$	$Trans(r)^{\mathcal{I}} \equiv r^{\mathcal{I}} \circ r^{\mathcal{I}} \subseteq r^{\mathcal{I}}$	
conjunction	$C \sqcap D$		$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	
disjuction	$C \sqcup D$		$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	$\mathcal{S}$
negation	$\neg C$		$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	
exists restriction	$\exists R.C$		$\{x \mid R^{\mathcal{I}}(x, C) \neq \varnothing\}$	
value restriction	$\forall R.C$		$\{x \mid R^{\mathcal{I}}(x, \neg C) = \varnothing\}$	
GCI		$C \sqsubseteq D$	$(C \sqsubseteq D)^{\mathcal{I}} \equiv C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$	
concept assertion		$(C(i))$	$(C(i))^{\mathcal{I}} \equiv i^{\mathcal{I}} \in C^{\mathcal{I}}$	
role assertion		$(r(i,j)$	$(r(i,j))^{\mathcal{I}} \equiv \langle i^{\mathcal{I}}, j^{\mathcal{I}} \rangle \in r^{\mathcal{I}}$	
role hierarchy		$R \sqsubseteq S$	$(R \sqsubseteq S)^{\mathcal{I}} \equiv R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$	$\mathcal{H}$
inverse roles	r^-		$(r^-)^{\mathcal{I}} = \{\langle y, x \rangle \mid \langle x, y \rangle \in r^{\mathcal{I}}\}$	$\mathcal{I}$
number restrictions	$\geqslant nR$		$\{x \mid \lvert R^{\mathcal{I}}(x, \top)\rvert \geq n\}$	$\mathcal{N}$
	$\leqslant nR$		$\{x \mid \lvert R^{\mathcal{I}}(x, \top)\rvert \leq n\}$	
qualifying number restrictions	$\geq nR.C$		$\{x \mid \lvert R^{\mathcal{I}}(x, C)\rvert \geq n\}$	$\mathcal{Q}$
	$\leq nR.C$		$\{x \mid \lvert R^{\mathcal{I}}(x, C)\rvert \leq n\}$	
nominals	$\{i\}$		$(\{i\})^{\mathcal{I}} = \{i^{\mathcal{I}}\}$	$\mathcal{O}$
functional roles		$Func(R)$	$Func(R)^{\mathcal{I}} \equiv ID(\Delta^{\mathcal{I}}, 2) \subseteq R^{\mathcal{I}} \circ (R^{\mathcal{I}})^{-1}$	$\mathcal{F}$

In general, the semantics of reasoning with ontologies applies the so-called open world assumption (OWA). In brief, the open world assumption means that a fact cannot be assumed to be false, only because it is not explicitly stated to be true. Instead, such facts are assumed to be unknown until contradictory evidence is found in the stated facts or derived from them. A second unexpected property for those used to UML models is that individual names are interpreted as pointers to the actual individuals. Two distinct individual names may refer to the same individual. In fact, special inequality axioms have to be added to an ontology, if this behavior should be avoided.

Eventually one should note that modeling with ontologies does not follow the frame-based paradigm [13]. A frame specification acts as a statement that something exists (a declaration) and it collects all relevant properties of the object in one place. For example, in UML a class declaration enumerates all relations of the class with other classes. Once the frame specification is finished, there is no way to alter the properties of the declared object, i.e. no relations can be added to the same class at a later time. In description logics, axioms are the fundamental modeling concept. Axioms 'talk' about objects but they do not declare them. If no axioms are given, an individual of a particular concept may have relationships with any other individuals. These relationships may be limited by axioms added in an importing ontology at a later time.

Rule Systems. As initially stated, rule systems add to the expressivity of ontology languages. Informally, class descriptions in OWL can only describe tree-like structures. It is not possible to model the concept uncle in 'the uncle y of x is the brother of z where x is also the child of z' because the induced graph with y, x, z as nodes and *uncle*, *brother* and *child* are relationship types is not a tree.

Generally, a rule has the form

$$A_1, A_2, \ldots, A_n \rightarrow D$$

and a rule system is a set of rules. The A_i are the rule conditions, the comma reads as a logical conjunction, and the D is the head. A rule can be processed with various semantics. The most frequently used ones are [1, 26]:

(Integrity) Constraint Rule. If the conditions $A_1, A_2, \ldots, A_n$ hold over some fixed domain of discourse, verify that the condition D also holds and output *true* or $false$.

Deductive Rule. If the conditions $A_1, A_2, \ldots, A_n$ hold over an initial domain of discourse, then verify the condition D also holds in the domain. If not, try to extend the domain such that D holds. Otherwise output $failure$.

Reactive Rule. If the conditions $A_1, A_2, \ldots, A_n$ hold over a domain of discourse then carry out action D (which may or may not change the domain).

Ontology reasoners that can also reason about rules apply the deductive rule semantics. Using a deductive rule, the above uncle concept can be easily modeled as

$$brother(y, x), child(x, z) \rightarrow uncle(y, z)$$

In an integrated ontology and rule system, rules cannot have arbitrary form, if decidability shall not be sacrificed. In fact, the above rule leads to undecidability. So called DL-safe rules avoid undecidability by restricting them to known individuals [20]. This means, every variable of a rule must occur in a non-DL concept in the rule conditions. It is possible to make the above rule DL-safe by adding three non-DL atoms

$$O(x), O(y), O(z), brother(y, x), child(x, z) \rightarrow uncle(y, x)$$

The O is an artifical non-DL concept. It contains the facts $O(i)$ for every individual $i \in N_I$. The atoms can be added automatically by the reasoner. Hence we do not explicitly specify the $O(x_i)$ in our examples. For practical applications this means that rules cannot be used to induce new individuals during reasoning as a DL $\exists$ concept description would do.

3.3 Ontology Specification Packages for Architecture Modeling

In this section we outline how ontologies can be used to support the 'consistency checked code approach'. Figure 8 displays the various ontologies that need to

be created and their roles with respect to the parts of an architecture model specification.

Every box in the figure represents a set of constraints over a shared vocabulary expressed using ontology language constructs. The `info` ontology takes the role of the concepts language – it introduces the architecture concepts and relationships and specifies the constraints for their composition. The `library.info` ontology represents the application architecture – it defines the individuals and relations by referring to the same concepts and relationship names as the `info` ontology. The `java` ontology is a necessary prerequisite for the specification of the platform mapping. It takes a similar role as the `info` ontology – it defines a schema but not that of the architecture but the schema of the platform. The `library.java` ontology represents a model of the code of the actually implemented system in terms of the platform concepts. Eventually, the `info2java` ontology specifies the mapping between concepts of the architecture and concepts offered by the platform.

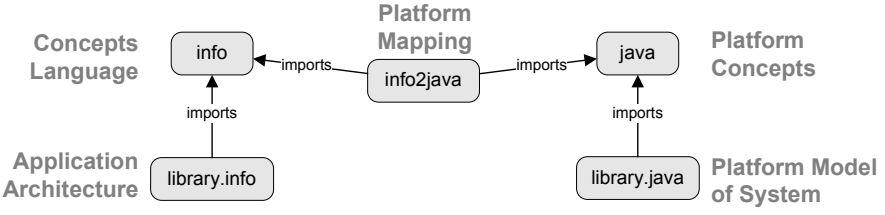


Fig. 8. Basic setup of ontologies to model the library system architecture and check the source code for consistency with the models

To conduct the various checks mentioned in section 1.2, the standard ontology reasoning services have to be invoked in different contexts as shown in figure 9. The `import`-relationship between two ontologies has the effect that all axioms of the imported ontology are also considered when a reasoning task is invoked for an importing ontology (union of statements).

Figure 9 shows how the initially stated architecture analyses (cf. section 1.2) can be implemented with the basic ontology setup from figure 8.

The following paragraphs briefly explain how ontology reasoners can be used to implement the required analyses. A more concrete illustration of an analysis of the library system introduced in chapter 2 (also cf. figure 6) will be given in chapter 4.

3.4 Concepts Consistency Analysis

The concepts consistency analysis translates straight forward into a *satisfiability check* invoked with the architecture concepts ontology (`info` in the library example). An ontology reasoner can be used to verify whether all concepts

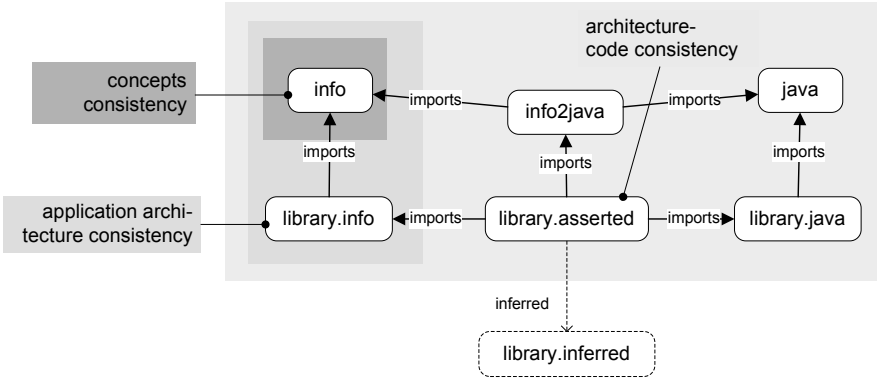


Fig. 9. Reasoning contexts for implementing several types of consistency checks

explicitly named in this ontology are satisfiable. In case they are not, a reasoner can

- output which concepts of an ontology are unsatisfiable
- output the axioms that are contradictory and, therefore, lead to unsatisfiability.

This information is a practically necessary feedback to the architect to enable him correcting his specification.

3.5 Application Architecture Consistency Analysis

This kind of analysis can also be covered by one of the standard ontology reasoning services. It can be implemented as a *consistency check* invoked with the `library.info` ontology. In this case, the architecture concepts ontology `info` contributes the TBox axioms and the application architecture ontology `library.info` provides the ABox assertions for the check. The open world semantics applied for reasoning also just meets what is required in this situation because the application architecture only incompletely specifies the system (see section 2.1). The concepts language of the library example requires that every Component provides at least one Interface. An application architecture specified by the single assertion

$$\mathsf{Component}(\mathit{users})$$

would be considered illegal using closed world reasoning because there is no specified Interface for the *users* component. With the open world assumptions, a reasoner just adds an anonymous individual such as

$$\mathsf{Interface}(\mathit{i1})$$

and continues drawing inferences. Contradictions can only occur, if inferred facts contradict one of the explicitly stated facts. As in the case of the concepts

consistency analysis, reasoners can provide feedback about which statements of the involved ontologies are inconsistent.

3.6 Architecture-Code Consistency Analysis

This type of analysis is more complex and requires some post-processing of the inferences produced by the reasoner. Given the setup in figure 9, the analysis starts with an invocation of a *realization* operation for the `library.asserted` ontology. This ontology represents the union of all other ontologies in the figure but does not necessarily contain additional axioms or assertions. Particularly, it unites axioms and rules defined in the platform mapping ontology `info2java` (a TBox) with the assertions that model the to-be-checked source code from the `library.java` ontology (the ABox). Applying the *realization* operation on this ontology has several effects:

- The mapping rules are applied by the reasoner. This leads to computing the $instance-of$ relationships between architecturally relevant elements of the platform and the architecture concepts from the `info` ontology. This means that the code structure is represented in terms of the architecture concepts language (inferred application architecture).
- The inferred 'implemented' application architecture is consistency checked against the TBox axioms of `info` using an open world assumption. This can detect unexpected relationships between source code elements that are not allowed according to the concepts language (provided appropriate axioms are given there). Given an appropriate axiomatization of the mapping rules, this step can also uncover unexpected source code elements which have no correspondence in the application architecture (e.g. a second LogicUnit within a Component would be such an unexpected element).
- The open world semantics can generate application architecture individuals (individuals with a $instance-of$ relation to a concept from the `info` ontology) which are not part of the specified application architecture. Equally, it can also generate platform individuals which logically follow from the asserted application architecture but which are missing in the actual code.

A reasoner can be asked to output the set of all inferred axioms and assertions in an additional `library.inferred` ontology. By post-processing this information we can achieve a closed world behavior and at the same time generate explanations about further inconsistencies with respect to such a semantics. We will given an example of how to use this information when we discuss analyzing the library example for architecture-code consistency (see section 4.5).

4 Modeling the Example with Ontology and Rule Languages

This section illustrates the models and analyses introduced earlier with the help of the library system.

4.1 Modeling the Concepts Language and Checking for Satisfiability

Figure 6 has shown the basic concepts for the library system. We can readily express these architectural concepts as axioms in an `info` ontology as shown in figure 8.

`info` Ontology
N_C := {Component, BusinessComponent, TechnicalComponent, Interface, IntInterface, ExtInterface, LogicUnit, Repository, Entity} N_R := {contains, provides, requires, use, name}
Axioms
(1) BusinessComponent $\sqsubseteq$ Component, TechnicalComponent $\sqsubseteq$ Component
(2) IntInterface $\sqsubseteq$ Interface, ExtInterface $\sqsubseteq$ Interface
(3) $\top \sqsubseteq \forall$name.`String`
(4) $InverseFunctional$(contains)
(5) Entity $\sqcup$ Repository $\sqcup$ LogicUnit $\sqsubseteq \forall$contains$^-$.BusinessComponent
(6) Repository $\sqsubseteq \exists$use.Entity
(7) Repository $\sqsubseteq \forall$use$^-$.LogicUnit
(8) Entity $\sqsubseteq \forall$use$^-$.(LogicUnit $\sqcup$ Repository)
(9) BusinessComponent $\sqsubseteq$ 1 contains.LogicUnit
(10) BusinessComponent $\sqcap$ TechnicalComponent $\sqsubseteq \bot$, LogicUnit $\sqcap$ Repository $\sqsubseteq \bot$, ...
(11) Component $\equiv$ TechnicalComponent $\sqcup$ BusinessComponent, ...

Fig. 10. Concepts language ontology for the library system

The ontology shown in figure 10 models the concepts language as depicted in figure 6. The concept and relationship names are given in the header of the figure (sets N_C, N_R). Restrictions on the use of concepts and relationship types can be expressed using either OWL axioms. Some of the relationships shown in figure 6 are mapped to specific OWL constructs. The extends relationship between concepts translates to a GCI axiom $\sqsubseteq$ between the OWL concept names (see lines 1 and 2 in figure 10).

The ontology is more precise than the initial figure 6 because it adds more accurate constraints to the concepts and relationship types. For example, the contains relationship type occurs between a BusinessComponent and Entity, Repository and LogicUnit. In general, there should only be one containing BusinessComponent for every Entity, Repository or LogicUnit. In OWL this property can be expressed

- by declaring contains to be an inverse functional property (see line 4 in figure 10).
- by limiting the 'parent' role of the contains relation to BusinessComponents (line 5)

The expressivity of OWL is very suitable for our purpose, because of the constructs to quantify participation in relationships and to derive ad-hoc relationships and concept sets when formulating axioms. Most axioms make quantified

statements about a single class and either state the constraints focusing on the subject (line 6: a repository 'must use some' Entity) or focusing on the object (line 7: a Repository or Entity 'may be used only by a' LogicUnit). According to our experience, these are very common type of architectural constraints found in practice.

The ontology of figure 10 is *satisfiable* and one may verify this property even without the help of an ontology reasoner. As an inconsistent example, suppose the architect reuses the `info` ontology and extends it into a `reportinfo` ontology (figure 11) with a concept ReportLogicUnit – units of functionality related to report creation. A ReportLogicUnit is considered a different concept than the already existing LogicUnit. Since a ReportLogicUnit requires access to some data to fill in reports he states that a ReportLogicUnit has to use a Repository which provides access to persistent Entities (line 1).

`reportinfo` Ontology `imports info`
$N_C := \{\mathsf{ReportLogicUnit}\}$
Axioms
(1) $\mathsf{ReportLogicUnit} \sqsubseteq \exists \mathsf{uses}.\mathsf{Repository}$ (2) $(\mathsf{ReportLogicUnit} \sqcap \mathsf{LogicUnit} \sqsubseteq \bot)$

Fig. 11. An unsatisfiable extension of the `info` ontology

When the `reportinfo` ontology is checked for satisfiability (for now without line 2), the reasoner reports that the ontology is consistent but also infers that the concept ReportLogicUnit is a subconcept of LogicUnit (written as an axiom: $\mathsf{ReportLogicUnit} \sqsubseteq \mathsf{LogicUnit}$). This is not what the architect intended. The problem can be solved by declaring ReportLogicUnit and LogicUnit to be disjoint (line 2). After this change, the ontology `reportinfo` is unsatisfiable, which signals to the architect that he made a modeling error.

This example is still very limited in size and the inconsistency might be spotted without the help of an ontology reasoner. However, for larger ontologies, possibly composed from different modules and having undergone several iterations, such analyses are certainly very valuable.

4.2 Modeling the Application Architecture and Checking for Consistency

Figure 12 states the ontology assertions to model the application architecture of the library system as depicted in figure 6. The assertions state that four components exist in the library system – three business components *users*, *catalogue* and *loans* and one technical component called *log*. Concrete relationships between component instances can be modeled with role assertions. For example, the *loans* component is expected to depend on the *users* and *catalogue* components in order to link its contained *loan* Entity with the borrowing Customer and the borrowed Medium.

`library.info` Ontology `imports info`
$N_I := \{users, catalogue, loans, log, loan,$ $user, copy, loans, medium, address, userRepo,$ $mediaRepo, userService, userAccess, logInterface, catalogueAccess\}$
Assertions
BusinessComponent($users$), BusinessComponent($catalogue$), BusinessComponent($loans$), TechnicalComponent(log)
Entity($loan$), Entity($user$), Entity($copy$), Entity($medium$), Entity($address$)
Repository($userRepo$), Repository($mediaRepo$), LogicUnit($userService$)
IntInterface($logInterface$), provides(log, $logInterface$)
IntInterface($userAccess$), provides($users$, $userAccess$)
IntInterface($catalogueAccess$), provides($catalogue$, $catalogueAccess$)
requires($loans$, $userAccess$), requires($loans$, $catalogueAccess$), requires($users$, $logInterface$), requires($catalogue$, $logInterface$)

Fig. 12. Instance ontology that defines the application architecture of the library system

As previously said checking the application architecture for consistency can be implemented by a standard *consistency check* provided by ontology reasoners. Due to the open world assumption, the consistency check will not fail when the application architecture is incompletely specified. In the example, line (9) from the `info` ontology demands that every BusinessComponent contains exactly one LogicUnit. On the other hand, the `library.info` ontology does not specify a LogicUnit conained within the *catalogue* BusinessComponent. As long as no other asserted constrains disallow it, the reasoner will infer a missing LogicUnit in an attempt to satisfy all axioms.

4.3 Modeling the Target Platform

To implement an ontology based architecture-code consistency check, a model of the code to be checked is needed as well. Similarly to the architecture concepts language, every platform provides concepts to implement a system. The difference it that these concepts are technology-driven while the architectural concepts are requirements-driven.

In case of the library system the Java platform is used. Hence, the relevant properties of Java source code need to be captured in a model. Figure 13 shows an ontology describing the structure of Java code. The ontology captures the major structural elements (Package, JavaClass, JavaInterface, JavaEnum, Method and attribute) along with their names (see jname property), and nesting (see parent relationship type) structure. Cross-references between elements are captured as either inheritance relationships (extends relationship type) or general dependencies (depends relationship type). To avoid undesired inferences at a later time we have to state that all concepts are pair wise disjoint (line (4) illustrates some of the axioms for enforcing disjunctness). Second, in case subconcept relationships are used it is sometimes necessary to state that a superconcept is the disjoint

`java` Ontology
$N_I := \{\mathsf{JavaType}, \mathsf{JavaClass}, \mathsf{JavaInterface}, \mathsf{JavaEnum}, \mathsf{Package}\}$ $N_R := \{\mathsf{parent}, \mathsf{depends}, \mathsf{extends}, \mathsf{jname}\}$
Axioms
(1) $\mathsf{JavaClass} \sqsubseteq \mathsf{JavaType}$, $\mathsf{JavaInterface} \sqsubseteq \mathsf{JavaType}$, $\mathsf{JavaEnum} \sqsubseteq \mathsf{JavaType}$ (2) $\mathsf{Method} \sqsubseteq \mathsf{Member}$, $\mathsf{Attribute} \sqsubseteq \mathsf{Member}$ (3) $\mathsf{JavaType} \equiv \mathsf{JavaClass} \sqcup \mathsf{JavaEnum} \sqcup \mathsf{JavaInterface}$ (4) $\mathsf{JavaClass} \sqcap \mathsf{JavaInterface} \sqsubseteq \bot$, $\mathsf{JavaClass} \sqcap \mathsf{JavaEnum} \sqsubseteq \bot$ $\mathsf{JavaEnum} \sqcap \mathsf{JavaInterface} \sqsubseteq \bot$, $\mathsf{Attribute} \sqcap \mathsf{Method} \sqsubseteq \bot \cdots$ (5) $Func(\mathsf{jname})$, $Func(\mathsf{parent})$ (6) $\mathsf{extends} \sqsubseteq \mathsf{depends}$

Fig. 13. Simple schema ontology for the Java Programming Language

union of its subconcepts. Without such a constraint the superconcept could be concluded to have additional individuals. Line (3) shows a so-called covering axiom for the subconcepts of JavaType to enforce this behavior. Line (5) states that the two properties parent and jname are functional, i.e. an individual representing a Java element can have at most one parent and at most one identifier.

The `java` ontology describes the schema of a Java program. To create an instance model, a Java parser can be used to create the ABox assertions according to the above schema. For example, the Java fragment

```
package library;

class CatalogueRepository extends AbstractRepository {
      Connection conn;
}
```

results in the following assertions:

$$\begin{aligned}&\mathsf{Package}(i_1), \mathsf{jname}(i_1, \texttt{"library"}), \mathsf{JavaClass}(i_2),\\ &\mathsf{jname}(i_2, \texttt{"CatalogueRepository"}), \mathsf{JavaClass}(i_3),\\ &\mathsf{jname}(i_3, \texttt{"AbstractRepository"}), \mathsf{extends}(i_2, i_3), \mathsf{parent}(i_2, i_1),\\ &\mathsf{JavaClass}(i_4), \mathsf{jname}(i_4, \texttt{"Connection"}), \mathsf{Attribute}(i_5),\\ &\mathsf{jname}(i_5, \texttt{"conn"}), \mathsf{parent}(i_5, i_2), \mathsf{depends}(i_5, i_4)\end{aligned}$$

4.4 Expressing the Platform Mapping

We are now in the position to formalize the platform mapping given in figure 7. The goal of the mapping is to describe how the individuals and relations expressed in a source ontology relate to individuals and relations expressed in a target ontology. In our case the source consists of the schema ontology `java` and the instance ontology `library.java`. The target schema ontology is `info`. The goal is to infer the $instance - of$ relationships of some individuals to the concepts named in the `library.info` ontology.

To express the mapping we use another ontology `info2java` which depends upon both schema ontologies (i.e. it includes `info` and `java`). We express the mappings using axioms and rules.

Mapping of Concepts. The strategy to express a mapping for a concept or set of concepts is

1. Define a concept name for every atomic property occurring in the mapping rule.
2. Formalize the property as either a axioms or DL-safe SWRL rules where the new concept occurs in the rule head (i.e. the rule will generate instances of the concept by adding *instanceof* relations)
3. State the necessary and sufficient criteria by relating the introduced concepts.

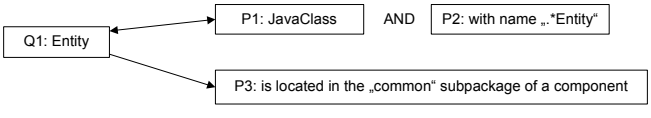
We will describe our approach with the help of the following example rule:

> An Entity maps to a JavaClass whose name ends with `XEntity` and which is located in the subpackage `common` of the component's package.

Each such mapping rule describes different properties the individuals must have, if they should be related. For example, in the destination `info` ontology the mapping rule requires an individual that is a member of the Entity architectural concept. In the source ontology `java` there are more criteria to be considered

1. Property 1: individuals that are members of the JavaClass concept
2. Property 2: individuals that have a jname property value that ends with `Entity`
3. Property 3: individuals that have a parent relationship to a Java package whose name is `common` and whose parent is a component's package.

For each property one has to decide whether it is a *necessary* or *sufficient* or *necessary and sufficient* condition for the mapping. The following figure illustrates the most likely correct interpretation of the example rule.



In the figure, the arrows indicate the logical implications that should apply. The following must hold

1. If something is an Entity, it must also be a member of the JavaClass concept and have a jname property value that ends with `Entity`
2. Conversely, if something is a member of the JavaClass concept and has a jname property value that ends with `Entity`, then it must also be an Entity.
3. If something is an Entity, it must also have a parent relationship to a Java package whose name is `common` and whose parent is a component's package.

The lower P3 property is only a *necessary* condition related to the Q1 property, because not any class inside a component's `common` subpackage should be an Entity.

Figure 14 shows the formalization of the example mapping rule about entities. The two properties P1 AND P2 and P3 are captured by the concepts MappedEntityByName and MappedEntityByLocation. The axiom (a2) states that being an member of MappedEntityByName is a *necessary and sufficient* condition for an individual to be a member of Entity (the $A \equiv B$ is equivalent to the two GCIs $A \sqsubseteq B$ and $B \sqsubseteq C$ and therefore expresses concept equality). The axiom (a3) states that being a member of MappedEntityByLocation is only a *necessary* condition for Entity.

The rule (r1) does the actual selection of appropriate Java classes from the code. The rule expresses that every JavaClass c with a jname cn and whose cn ends with the string `Entity`, is inferred to be a member of the MappedEntityByName concept. In the rule language SWRL basic functions such as the shown testing for string suffixes are available. By the axiom (a2) the concept MappedEntityByName is equal to the concept Entity which allows a reasoner to conclude which individuals originally asserted in the `library.java` ontology are also a member of the architectural Entity concept (i.e. such individuals belong to two concepts at the same time).

`info2java` Ontology imports `info`, `java`
$N_I :=$ {MappedEntityByName, MappedEntityByLocation PackageByNameCommon} $N_R := \{\}$
Axioms
(a1) MappedEntityByLocation $\sqsubseteq$ JavaClass $\sqcap$ $\forall$parent.(PackageByNameCommon$\sqcap$ $\forall$parent.Component) (a2) Entity $\equiv$ MappedEntityByName, MappedEntityByName $\sqsubseteq$ JavaClass (a3) Entity $\sqsubseteq$ MappedEntityByLocation
Rules
(r1) $JavaClass(c), jname(c, cn), swrl : endsWith(cn, \texttt{"Entity"})$ $\rightarrow MappedEntityByName(c), name(c, cn)$ (r2) $Package(p), jname(p, \texttt{"common"}) \rightarrow PackageByNameCommon(p)$

Fig. 14. Formalized Mapping for Entity

According to this scheme, mappings for all architectural concepts can be formalized. The properties to select individuals that take part in a mapping depend on the information present in the code models. Types of rules to link architectural information to source code elements commonly found in practice are:

- *Naming conventions*: all entities (in the sense of an architectural concept) are implemented as a single Java class with the name suffix `Entity`.
- *Base classes or interfaces*: all entities implement the `IEntity` interface

– *Location conventions*: all entities of a component reside in the subpackage `common.entities` within the component package.
– *Java annotation*: all entity classes are marked with an `@Entity` annotation.

Mapping of Relationship Types. Besides mapping of individuals, we also have to define and formalize mappings for relationships, because only then we can eventually check how stated architectural relationship constraints are fulfilled in the code. The basic idea is to translate code-level relations into architectural relations by stating the conditions of such mappings as rules. Figure 15 shows two such rules for inferring the architectural use and provides relationships from the code. This formalization follows the natural language description of the mapping shown in figure 7.

Rule (r1) simply states that individuals related by an asserted or inferred depends relationship are also related by a use relationship. The rule selects all individuals related by a depends relationship from the `library.java` ontology. Because the mapping for concepts 'reuses' the individuals asserted in the `library.java` ontology to also express their membership with concepts from `library.info`, it is sufficient to add a relation assertion for the same individuals x and y. Rule (r2) illustrates a conditional mapping of a relationship. The provides relation is only inferred for two individuals c and i, if

– c is a Component (for which we know it is mapped to a Java Package),
– i is an Interface (whch is also mapped to a Java Package)
– i's parent is *comm* and *comm*'s parent is c and *comm* is a Package named `common` (see figure 14)

`info2java` Ontology (continued)
Rules
(r1) $depends(x, y) \rightarrow use(x, y)$
(r2) $Component(c), Interface(i), parent(i, comm),$ $PackageByNameCommon(comm), parent(comm, c) \rightarrow provides(c, i)$

Fig. 15. Formalized Mapping for relationship types use and provides

4.5 Checking for Architecture-Code Consistency

With the previous mapping formalization we are eventually in the position to verify the correctness of the code with respect to architectural constraints. We illustrate two basic kinds of constraints stated in the concepts language ontology `info` together with code-level assertions from `library.java` that eventually violate the constraints.

The first type of constraint is of the form

(1) A concept X can only be related to a concept Y with a relation r.

where X, Y and r are constrained in the concepts language ontology. In the library example, the axioms in lines (5) and (7) from the `info` ontology (cf. figure 10)

belong to this type. With the shown mapping of Entity and a similar mapping for Repository the Java fragment

```
class MediumEntity {
  MediumRepository rep;
}
```

would be inconsistent because a relationship $\mathsf{use}(e, r)$ would be inferred (with e being the Entity called `MediumEntity` and r being the Repository called `MediumRepository`). This relationship violates the axiom in line (7)

$$\mathsf{Repository} \sqsubseteq \forall \mathsf{use}^{-}.\mathsf{LogicUnit}$$

from the `info` ontology. The only way the reasoner could satisfy this axiom is to infer $\mathsf{LogicUnit}(r)$ as well which on the other hand contradicts line (10):

$$\mathsf{LogicUnit} \sqcap \mathsf{Repository} \sqsubseteq \bot$$

The concepts of the `info` ontology have been declared to be disjoint to avoid such unintended inferences.

The second type of constraint is of the form

(2) A concept X must be related to a concept Y with a relation r.

In the example `info` ontology, the line (6) states such a constraint – a Repository must use at least one Entity. What happens for a $\mathsf{Repository}(r)$ inferred from the source code which does not fulfill this constraint? The *realization* operation invoked with the `library.asserted` ontology will create an anonymous individual $\mathsf{Entity}(\_\ e)$ asserted in the `library.inferred` ontology. Through the mapping axioms (a2) (cf. figure 14)

$$\mathsf{Entity} \equiv \mathsf{MappedEntityByName},\ \mathsf{MappedEntityByName} \sqsubseteq \mathsf{JavaClass}$$

the reasoner also concludes the assertions $\mathsf{MappedEntityByName}(\_e)$ and $\mathsf{JavaClass}(\_\ e)$. This new individual $\_\ e$ occurs because of the open world assumption. These inferences are problematic in this case because we would actually expect an output of the type 'an Entity for the Repository r is missing in the code'. We can deal with this problem in two ways:

1. Restrict the set of Entity individuals to the ones present in the code
2. Compute a difference between the asserted individuals and relations in the `library.java` ontology and the additionally inferred ones in the `library.inferred` ontology.

Option 1 can be achieved by adding a general inclusion axiom with a nominal expression (explicit enumerations of the members of a concept) on the right hand side as follows:

$$\mathsf{Entity} \sqsubseteq \{i_1, i_2, i_3, \ldots i_n\}$$

With such an addition (e.g. to the `library.info` ontology) the reasoner cannot infer $\mathsf{Entity}(\_\ e)$ and therefore outputs the entire `library.asserted` ontology to be inconsistent.

Option 2 can be easily implemented as a post-processing step after *realization*. It is easy to select the inferred assertions of individuals that are not already present in the `library.java` ontology. To output only missing code elements the search should be limited to individuals that are inferred members of any of the platform concepts.

5 Related Work

The idea of analyzing existing source code for consistency with a given architecture model is not new and goes back to the reflexion models described by Murphy et al. in 1995 [22]. In the method, the architect defines a high level model representing the architecturally relevant modules and the assumed dependencies between those modules. For each high-level module, a mapping defines with source code elements correspond to this module. Based on this information, a tool extracts the actual dependency graph from the source code and presents the differences with the defined module dependency graph.

This basic idea has been refined over the years resulting in research prototypes and commercial tools [22, 17, 7, 27, 4, 34]. All of these tools focus on the identification of illegal dependencies between source code elements, based on a specification of the planned dependencies on a higher abstraction level. They do not consider checking for existence of elements and they do not include the checking of properties (such as naming conventions captured in our mapping rules).

A fundamental problem of all the named approaches is that the architect is very limited in specifying the planned architecture. All tools either force the architect to specify dependency constraints either directly on the code level (which provides no abstraction) or the architect has to express this architecture in terms of a fixed concepts language, which however misses the point that such a language must be adaptable to the specific requirements. Therefore, often architects cannot express their architectures in a natural form or cannot express certain constraints at all or the maintenance of the constraints is effortful.

As an example, consider the problem of mapping the architectural concepts of the library system (see figure 6) to the concept language shown in figure 16. We could opt to map our concept Component to SonarJ's concept of a Slice and express some of the Component-internal structures with the help of SonarJ's Layers. A reasonable choice would be to introduce the two layers *businesslogic* and *persistence* and assign Entities and Repositories to the persistence layer. SonarJ's default semantics of layers would provide us with the desired effect that which this choice every piece of the businesslogic-layer can access the entities but not vice versa. Although for many systems this rough translation of concepts may succeed, this is not an ideal solution. The main disadvantage is that the original concepts language is no longer visible in the tool. Analysis results cannot be reported in the original concepts language.

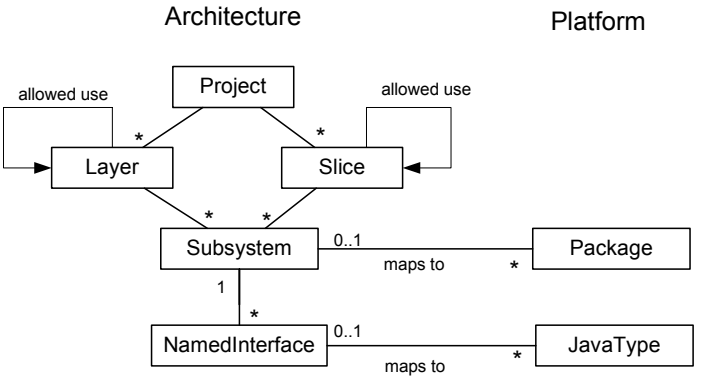


Fig. 16. Concepts language and possible platform mappings of the tool SonarJ [34]

Besides this fundamental problem, there are a number of technical problems

- The tools require the architect to thoroughly define all elements of the application architecture and their mapping to the source code. This means, whenever new elements are added to the source code, the architecture specification and the platform mapping in the tool have to be extended in order to include the new elements in the analysis. It would be desirable to let a tool discover the instances of a specific concept from the code, and have the rules associated with the concept applied automatically. This is particularly useful for concepts whose instances are not completely specified in the application architecture.
- The tools focus on checking the consistency of the dependencies alone. However, a platform mapping also includes storage and naming conventions such as those shown in figure 7.

6 Conclusion

In this article we have worked out the 'consistency checked code approach' as a practical option to implement an architecture centric development process. Compared to 'separate models and code' it prevents the drift between architecture and code by regular analysis and correction cycles. Compared to generative model driven development, this approach is more lightweight as is does not require a comprehensive language infrastructure. It is also suitable for less mature domains where it is difficult to design a detailed architecture initially and it particularly applicable to existing software systems as a support during restructuring activities.

Second, we have discussed and demonstrated with the help of an example that the structural specification of a software architecture includes a domain specific concepts language, an application architecture and a platform mapping. We have derived requirements to implement the 'consistency checked approach' from this

understanding of a software architecture. In our view, particularly important requirements are

1. the comprehensive support of different types of analyses
2. a lightweight infrastructure to specify and evolve architectural models,
3. and support for modularity and composition.

The major message of our paper is that ontologies and rule systems can be used as a design language for software architectures (items 2 and 3 in the list). We further argue and demonstrate with the help of an example that an ontology and rule system reasoner provides the necessary analyses (item 1).

More specifically a reasoner can be used to check the satisfiability of the architecture concepts language, the consistency of the application architecture with respect to the concepts language and it can be used to implement architecture-code consistency checks. We have also given a few examples for how each analysis can identify errors in the architecture specification and the code.

Compared to other architecture-code consistency checking methods such as those described in [22, 17, 7, 27, 4, 34] our method has the following benefits

- It allows the architect to directly describe a software architecture in a structured way in suitable models. There is no need to manually translate architectural constraints into a predefined metamodel or to the code level to enable architecture-code consistency analyses.
- It integrates previously distinct kinds of analyses (architectural conformance of dependencies, conformance with rules for code organisation, conformance with some low level programming style guide rules such as naming conventions) into a common framework. This enables us to express and verify interactions between different properties.
- It prevents the architect from specifying unsatisfiable models
- It supports modular specifications, reuse and composition of specifications.

Admittedly and mostly for space reasons we have not thoroughly discussed limitations and drawbacks. Some of the issues we see are

- Ontologies and ontology-integrated rule systems so far only support so-called monotic reasoning. This and the open world assumption are reasons why negative facts cannot be expressed easily. For example, we cannot state 'a TechnicalComponent cannot use a BusinessComponent'. Instead we have to put this into a positive form and enumerate all concepts a TechnicalComponent can *only* use. This is particularly problematical if the family of component types shall be extended in the future.
- The open world assumption and the property that an individual can belong to several concepts, sometimes require lengthy and numerous axioms that just prevent unintended inferences, but not model architecture. The disjointness and covering axioms in the `java` ontology are examples of such axioms.

– We have not discussed how larger grain concepts, their mapping and architecture-code consistency checks can be modeled and analyzed using ontologies. For example our Component concept maps to a set of Java classes contained in a package. To infer the use relationships of a Component, the depends relationships between classes would have to be lifted to the Java package level. This in turn would require a transitive parent role in the java ontology. Unfortunately, the use of transitive roles badly interacts with other description logic constructs [21] and therefore further limits expressivity.

The procedure we have described is implemented as a prototype tool built around the Pellet OWL reasoner [29] and the Protégé ontology workbench [31]. In their latest versions, both support OWL 2 and reasoning with DL-safe rules encoded in SWRL. We plan to extend our research as follows: The validation of the approach in realistic settings is still an open issue. We intend to investigate whether the expressivity provided by OWL 2 and SWRL are sufficient to capture the structural properties of industrial-strength architectures. We also expect the performance of inference services to become more critical as the size of the evaluated systems increases. In order to enable the actual use of our approach we also like to integrate the described procedure as a plugin for the Eclipse integrated development environment.

References

[1] Antoniou, G., Van Harmelen, F.: A Semantic Web Primer (Cooperativer Information Systems). MIT Press, Cambridge (2004)
[2] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambirdge University Press, New York (2003)
[3] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading (2003)
[4] Becker-Pechau, P., Karstens, B., Lilienthal, C.: Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln. In: Gruhn, V., Biel, B., Book, M. (eds.) Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), pp. 27–38 (2006)
[5] Bennicke, M., Baresel, A.: Establishing efficient code quality reviews to monitor external projects. In: Proceedings of International Conference on Maintenance Engineering (ICME), Chengdu, China (2006)
[6] Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web: a new form of web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American 284(5), 34–43 (2001)
[7] Bischofberger, W., Kühl, J., Löffler, S.: Sotograph - a pragmatic approach to source code architecture conformance checking. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 1–9. Springer, Heidelberg (2004)
[8] Bosch, J.: Design & Use of Software Architectures. Addison-Wesley, Reading (2000)
[9] Brooks, F.P.: The Mythical Man-Month, anniversary edition. Addison-Wesley, Reading, (1995)

[10] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, Reading (2002)
[11] European commission research project. Marrying Ontology and Software Technology (MOST) (2009), Project Homepage: http://www.most-project.eu/
[12] Favre, J.-M.: Languages evolve too! changing the software time scale. In: Eighth International Workshop on Principles of Software Evolution (IWPSE 2005) (2005)
[13] Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P.F., Sattler, U.: Owl 2: The next step for owl. Journal of Web Semantics 6(4), 309–322 (2008)
[14] Horrocks, I., Sattler, U., Tobies, S.: Practical reasoning for very expressive description logics. Logic Journal of the IGPL 8, 239–264 (2000)
[15] ISO/IEC/(IEEE). ISO/IEC 42010: International Standard Systems and software engineering - Recommended practice for architectural description of software-intensive systems (2007)
[16] Jouault, F., Kurtev, I.: Transforming models with atl. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
[17] Koschke, R., Simon, D.: Hierarchical reflexion models. In: Proc. of 10th Working Conference on Reverse Engineering (WCRE 2003), pp. 36–45. IEEE Computer Society, Los Alamitos (2003)
[18] Kruchten, P.: The 4+1 view model of architecture. IEEE Software 12(6), 42–50 (1995)
[19] Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: Owl 2 web ontology language profiles (October 2009), http://www.w3.org/TR/owl2-profiles/
[20] Motik, B., Sattler, U., Studer, R.: Query answering for owl-dl with rules. Journal of Web Semantics: Science, Services and Agents on the World Wide Web 3(1), 41–60 (2005)
[21] Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. Journal of Artificial Intelligence Research 36, 165–228 (2009)
[22] Murphy, G.C., Notkin, D., Sullivan, K.: Software reflexion models. bridging the gap between source and high-level models. In: Proc. of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 18–28. ACM Press, New York (1995)
[23] Object Management Group: MDA Guide V1.0.1 (2003)
[24] Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (July 2007)
[25] OpenGALEN Foundation: Opengalen website, http://www.opengalen.org (last visited September 2010)
[26] Parreiras, F.S., Staab, S., Winter, A.: On marrying ontological and metamodeling technical spaces. In: Foundations of Software Engineering, The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers (2007)
[27] Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software archi-tecture. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 167–176. ACM Press, New York (2005)
[28] Siedersleben, J.: Moderne Software-Architektur. dpunkt.verlag (2004)
[29] Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical owl-dl reasoner. Web Semantics: Science, Services and Agents on the World Wide Web In Software Engineering and the Semantic Web 5(2), 51–53 (2007)
[30] TopQuadrant, Inc. Topbraid vendor website, http://www.topquadrant.com (last visited September 2010)

[31] Stanford University. The protégé ontology editor and knowledge acquisition system, http://protege.stanford.edu/ (last visited September 2010)
[32] Völter, M.: Software architecture: A pattern language for building sustainable software architectures (March 2005), http://www.voelter.de/data/pub/ArchitecturePatterns.pdf (last visited September 2010)
[33] Völter, M.: Architecture as a language. IEEE Software 27(2), 56–64 (2010)
[34] von Zitzewitz, A.: Erosion vorbeugen. Javamagazin, 28–33 (February 2007)

Using Role-Play Diagrams to Improve Scenario Role-Play

Jürgen Börstler

Umeå University, Sweden

Abstract. CRC-cards are a common lightweight approach to collaborative object-oriented analysis and design. They have been adopted by many educators and trainers to teach object-oriented modelling. In our experience, we have noticed many subtle problems and issues that have largely gone unnoticed in the literature. Two of the major issues are related to the CRC-card role-play as described in the literature. Although CRC-cards are representing classes, they are also utilized as substitutes for the actual objects during the scenario role-play. Furthermore, it is quite difficult to document or trace the scenario role-play. We propose using Role-Play Diagrams (RPDs) to overcome these problems. Our experience so far is quite positive. Novices have fewer problems with role-play activities when using these diagrams. Teaching and learning the new type of diagram adds only little overhead to the overall CRC-approach. Although our improvements specifically target the teaching and learning of object-oriented modelling, we believe that RPDs can be successfully applied in professional software development.

1 Introduction

The usage of an object-oriented language does not in itself guarantee that the software developed will actually be object-oriented. Employing an appropriate modelling approach, supported by suitable tools, will support the development of object-oriented systems.

CRC-cards [1,2] are an informal yet powerful tool for supporting object-oriented modelling that was developed as a tool for teaching object-oriented thinking to (experienced) programmers [1]. Many educators and trainers have adopted CRC-cards as an approach to teach object-oriented technology and convey a bigger picture of (systematic) objected-oriented development [3,4,5,6,7,8]. CRC-cards are also used widely outside the educational context, in particular to support responsibility driven design approaches [2,9,10,11] or architecture evaluation [12]. They are also used in areas not directly related to software development, like for example product modelling and maintenance [13,14] or information behavior studies [15].

The power of the CRC-card approach lies in its associated scenario role-play activities that support collaborative design and active collaborative learning [16]. Role-playing scenarios is an effective way to simulate or explore hypothetical situations [17]. The role-play participants are assigned roles they enact according to a predefined scenario, much as actors following a script when playing the

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 309–334, 2010.

characters in play. In object-oriented development, the characters are the objects in a (software) system and the scenarios are hypothetical but concrete situations of system usage. During the role-play the participants learn a lot about the roles they play. The interactivity of the role-playing supports creativity and sharing of knowledge. Since the roles and scenarios can be easily varied, it is possible to explore many design alternatives without actually building any prototypes.

The remainder of this paper is organized as follows. After a brief discussion of object-oriented modelling, we introduce CRC-cards (Section 3) and a new type of diagram, called Role-Play Diagrams (RPDs, Section 4). In Section 5–6, we describe an analysis and design process that utilizes CRC-Cards and RPDs. Section 7 presents an example step-by-step, followed by a brief discussion of possible improvements to our approach. Our experience from using CRC-cards together with role-play diagrams in our introductory programming courses is summarized in Section 9.

2 Modelling Object-Oriented Software

The goal of modelling is to develop an as simple as possible model of a problem or domain that still correctly reflects all relevant aspects we are interested in. That makes it easier to focus on the essential properties and phenomena, without being distracted by irrelevant details or aspects [18].

The modelled software objects will frequently have real world counterparts in the problem domain. This can make modelling a straightforward activity, but also lead to some confusion. The modelled software objects should not be confused with their real world counterparts. Although the software objects will often reflect properties of their real world counterparts, will also have properties that their real world counterparts never can have. A physical book for example is removed from the library, when it is checked out. A book object in a (software) model however stays in the library and is only marked as on loan. In a "real world" library, we would never make the borrowers responsible for keeping track of their unpaid overdue fines. In a (software) model however, this might be a good design choice, since trust is no issue there.

It should also be noted that there are several levels of modelling. Aspects, properties, or details that are irrelevant during analysis might very well be important during design or implementation. There usually is no single correct model. Real world aspects can be modelled quite differently, depending on the actual problem. People for example will be modelled quite differently in a course registration system for a university compared to a journal system for a hospital.

3 CRC-Cards

CRC stands for **C**lass, **R**esponsibilities and **C**ollaborators [19][1]. A CRC-card is a standard index card that has been divided into regions, as shown in figure 1.

[1] **C**andidates, **R**esponsibilities and **C**ollaborators according to [10] and **C**lass, **R**esponsibility, **C**ollaboration according to Beck and Cunningham's original paper [1].

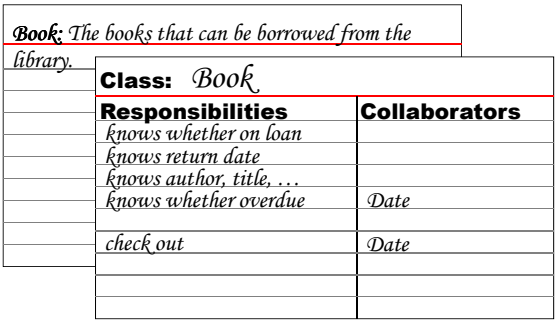


Fig. 1. Example CRC-card (front, at top, and back) for a `Book` class in a library system

A CRC-card corresponds to a **class**, i.e. a description of objects of the same kind. The objects are the things of interest in the problem or application domain. The class name is written across the top of the CRC-card. A class should have a single and well-defined purpose that can be described briefly and clearly. It should be named by a noun, noun phrase, or adjective that adequately describes the abstraction.

A **responsibility** is a service the objects of a class provide for other objects. A responsibility can be to know something or to do something. A book object in a library might, for example, be responsible for checking itself out and knowing its title and due date (see Figure 1). The responsibilities of (the objects of) a class are written along the left side of the card.

A **collaborator** is an object of another class "helping" to fulfil a specific responsibility. A collaborator can for example provide further information, or take over parts of the original responsibility (by means of its own responsibilities). A book object, for example, can only know whether it is overdue, if it also knows the current date (in Figure 1 this information is provided by a collaborator object of class Date). The class names of collaborators are written along the right side of the card in the same row as the corresponding responsibility.

The back of the card can be used for a brief description of the class' purpose, comments and miscellaneous details.

CRC-cards are particularly well-suited for collaborative design and active learning. It forces participants to reason about models and explain design decisions to their peers. It also enables participants to reflect on their modelling activities. This helps participants to build up a common vocabulary and understanding of the problem and possible solutions (and the modelling process as well).

A group size of 4–6 people seems to work best. Smaller groups usually lack the right blend of different backgrounds. In larger groups, it becomes too difficult to reach a consensus. It is important not to get bogged down in discussions about implementation details. The goal is to develop, discuss, evaluate and test different object-oriented models.

One goal with CRC-card usage is "to immerse the learner in the 'object-ness' of the material" [1, p 1]. The CRC approach is an anthropomorphic approach. Objects are interpreted as living entities that can act on their own behalf. Objects take a self-centered view. It should not be necessary to ask a third party object about things a particular object naturally should know by itself. This object-as-person metaphor [20] supports the kind of object thinking we want to teach our students. However, it is very important to carefully distinguish the properties of the model objects from the properties of their "real world" counterparts, as discussed above.

4 Role-Play Diagrams

Common role-play approaches use the CRC-cards as the "characters" in the role-play [2]. This gives the CRC-cards a double role: they represent classes and "stand in" for the actual objects [1]. This can be very confusing for novices, since the concepts of class and object are not clearly distinguished. We therefore developed role-play diagrams (RPDs) to reinforce this distinction. RPDs furthermore provide excellent support for documentation of the role-play. They can also easily be used to keep track of the role-play as it unfolds and support recovery of the latest consistent state of the role-play.

Role-Play Diagrams (RPDs) are used to document object interaction. The objects in a RPD are instances of the classes modelled by CRC-cards. The RPD is a new type of diagram that covers the most important aspects from UML object and collaboration diagrams [21]. However, RPDs are simpler and less formal than their UML counterparts and therefore better suited for recording scenarios in parallel to the role-play activities. Our experience has shown that drawing sequence diagrams "on the fly" is infeasible. Their prescribed structure makes them difficult to change and extend. Sequence diagrams also lack structural information, which is quite important for describing a scenario's state in detail.

Objects in RPDs are represented by object cards. An object card is an instance of a CRC-card showing its name, class and properties relevant for the current scenario. In Figure 2, for example, we have an object `aBook`, which is an instance of our CRC-card (class) `Book` from Figure 1. Usually, we use large post-it notes for the object cards and develop the RPD successively on the whiteboard or a large sheet of paper.

Objects that "know" each other (i.e. are in the same scope) are connected by a line. Two objects `a:A` and `b:B` can, however, only be connected when (at least) one of the corresponding CRC-cards for `A` and `B` lists the other as a collaborator. During the role-play, communication is only possible between connected object cards. Please note that being collaborators only means that there can be communication. In an actual scenario, communication paths must often first be established explicitly, before connecting lines can be drawn. Requests (messages sent) are documented on the connecting lines between the communicating objects. A request corresponds to an actual service requested by an object and must correspond to a responsibility listed on the CRC-card corresponding to the serving object.

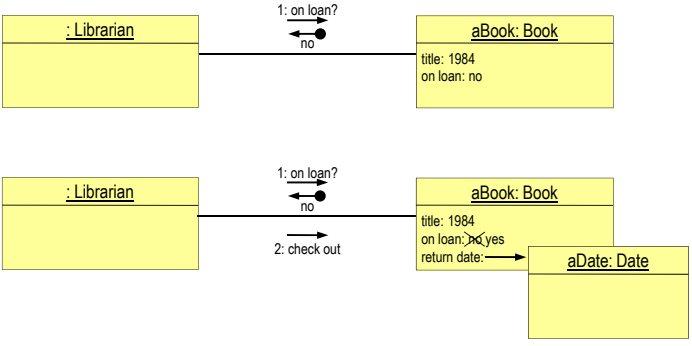


Fig. 2. Steps of a scenario "borrowing book '1984"' (slightly simplified)

In Figure 2 the requests `on loan?` and `check out` correspond to `Book`'s responsibilities `knows whether on loan` and `check out`, respectively (see Figure 1). A small arrow denotes the direction of a request[2]. A simple numbering scheme is used to keep track of the ordering of requests. Requests can be annotated with data flow for example to document information that is returned.

We do not prescribe any specific notation for the parameters of requests or the object properties. Our main design goal was simplicity. The usage of RPDs should add as little as possible overhead to the role-playing.

As soon as the knowledge of an object changes, the corresponding object card is updated accordingly. In Figure 2 the request `on loan?` returns a `no` and after checking out `aBook` its property `on loan` is changed to `yes`. Furthermore, we add a new `Date` object, since `Book` objects also have a responsibility `knows return date`. Please note that this was only possible, since `Date` is a collaborator for responsibility `check out`. At the end of a scenario role-play, the RPD documents exactly what happened, including relevant state changes. If necessary, this information could be easily translated into a UML sequence diagram, for example.

5 The CRC/RPD-Process: Object-Oriented Analysis

The CRC-card approach is particularly well suited, when the problem is not well defined. During Object-Oriented Analysis, the problem and application domains are analysed to understand the problem at hand and identify the things that actually should be done and those that are outside the system that should be developed.

A typical process includes the following steps [2,5]:

- identify candidate classes (see Section 5.1),
- filter the list of candidates (see Section 5.2),

[2] This particular notation is borrowed from the Booch notation's object diagrams [22], that did not make it into the UML standard.

- create CRC-cards for the remaining candidates (see Section 5.3),
- allocate responsibilities to CRC-cards/classes (see Section 5.4),
- define scenarios to test/evaluate the model (see Section 5.5),
- prepare the scenario role-play (see Section 5.6),
- perform the actual role-play (see Section 5.7),
- record scenarios (see Section 5.8), and
- update CRC-cards and scenarios (see Section 5.9).

Please note that these steps are not performed in strict sequence. Discussing responsibilities, for example, needs to be done to make informed decisions about the meaningfulness of candidate classes. The last three steps are always performed in parallel.

5.1 Identify Candidate Classes

The purpose of this first step is to generate a list of candidate classes that might be of interest for the problem at hand; classes that potentially take "responsibility" for some part of the functionality the system. Good candidates should have properties and behaviour. If objects differ only in the values of their properties, we consider them as objects of the same class. All books in a library for example would be considered as objects of the class `Book` (see Figure 1).

Classes should be named by a noun, noun phrase or adjective in singular form. Anything that cannot be named given these restrictions is most probably not a suitable candidate for a class (but maybe a responsibility).

An initial list of candidates can be generated in a brainstorming session. A simple approach to support this step is the *noun extraction approach*; to underline all nouns and noun phrases in the problem statement. One should keep in mind that brainstorming is an idea-generating technique, not an idea-evaluating technique. During the brainstorming session there should be little or no discussion on the suitability of candidate classes. All candidates should be recorded. Discussions should be deferred until the next step (Filter Candidates). It is recommended to take turns (give every voice a chance) and collect all candidates visibly for all participants for later evaluation.

5.2 Filter Candidates

The goal of this step is to cut down the number of candidates to a manageable size, preferably no more than ten. All candidates irrelevant for the problem at hand are discarded. According to Wirfs-Brock and Kean, "[c]andidates generally represent work performed by your software, things your software affects, information, control and decision making, ways to structure and arrange groups of objects, and representations of things in the world that your software needs to know something about" [10, p 106]. It should be noted that it is not necessary

to find all candidates at once. Further candidates will likely be uncovered during later steps.

The following guidelines can be useful for the filtering process:

- Merge synonyms into the candidate with the most suitable name.
- Merge candidates with largely overlapping responsibilities; if possible into a single abstraction that covers all candidates. In a library system, for example, there might be different kinds of entities that can be borrowed, like books, music, films, and software. If these are handled in exactly the same way by the system, it might be better to merge them into a new candidate, like for example "Lendable" and treat them as of the same kind (the original candidates might also be subclasses of the new common candidate).
- Discard candidates that cannot be properly named by a noun, noun phrase, or adjective. One should look particularly for noun-verbs describing services. Such candidates are often responsibilities of other candidates.
- Discard candidates that seem insignificant or vague, especially if their purpose cannot be described properly.
- Discard candidates that describe user interfaces or implementation details. Usually such details are not important to understand the problem. Considering them too early might constrain our solution space.
- Discard candidates that model (simple) properties of other candidates.
- Discard candidates that do not have any responsibilities.
- Discard candidates that are outside the system, like users or external systems. They are only needed, if the system's behaviour depends on their properties.
- Discard candidates that are outside the actual problem space. Candidates should only be based on actual or highly probable requirements. Although it might be important to foresee changes, one should focus on the problem at hand. Otherwise the model might become unnecessarily complex or even solve the wrong problem.
- Discard candidates that refer to the system itself, like "the system" or "the application".
- Discard candidates that represent objects, but make sure there is a candidate representing a corresponding class. Such a class is needed even when there only is a single instance (Singleton).

5.3 Create CRC-Cards

For each remaining candidate class a CRC-card is produced, according to the description in Section 3. Class names should be chosen carefully. A good name is specific and descriptive and can therefore easily be associated with a class' responsibilities. In a library system, for example, `Borrower` is a much better class name for the library end users than User, Person, or Client.

The back of the card should be used to briefly and clearly describe the purpose of the class to make sure that the class name is interpreted correctly. If this is not possible using a sentence or two, the class' purpose might be unclear or it

has too many of them. In the latter case, it should be split up to get classes with single unambiguous purposes.

5.4 Allocate Responsibilities

Responsibilities belong to the service providers and are listed on the CRC-cards that provide the services. Responsibilities should be evenly distributed among classes. There should be no single class responsible for all knowledge or all behaviour, so-called data or behavioural "god classes" [23].

Some responsibilities are obvious from the problem description or application domain. In a library system, for example, there need to be a responsibility for lending books. It seems therefore sensible to add a responsibility `check out` to the `Book` card, since books have some knowledge relevant for this responsibility (`knows whether on loan`, see Figure 1). Further responsibilities could be found by looking for verbs or adjectives in the problem description, similar to the noun extraction approach for finding candidate classes. Again, it is not at all necessary to identify all responsibilities up front. It is sufficient to identify a number of key responsibilities sufficient for a meaningful scenario role-play.

"[R]esponsibility denotes both duty and power ... an object should fulfil its obligations, and should have the ability to accomplish them" [3, p 203]. For each responsibility, it must be considered whether the objects need collaborators to fulfil their obligations. If collaborators are needed, their names must be entered on the CRC-card. This "delegation" of sub-responsibilities might also imply new responsibilities for the collaborators and their collaborators, etc. Quite often, several classes have similar or even identical responsibilities. This is a sign for close collaboration. However, there should not be too many overlapping responsibilities; in such cases classes should be merged (see Section 5.2). On the other hand, one must also avoid having a single class responsible for everything important in the model. In a library system, for example, borrowing a book is a quite complex responsibility that will likely require close collaboration of many objects.

At this stage it is very important to avoid discussing implementation details. They might lead to premature design decisions that prevent the team from identifying alternative distributions of responsibilities.

5.5 Define Scenarios

Scenarios are the test cases for the CRC-card model. Each scenario represents a specific example of system usage. It is like a script in a play that is enacted by the objects defined in the CRC-card model. When enacting a scenario the team follows the responsibilities and collaborations defined on the CRC-cards to validate whether the model can handle this particular test case.

Scenarios must be concrete and clearly defined. The combinatorial explosion of possible decisions will otherwise make the role-playing unmanageable. A scenario like *A user borrows a book* involves numerous special cases, like the following:

the borrower might not be user or not allowed to borrow books for some other reason; the library might not have the book or if it has it might be on loan; how will borrowing affect the "state" of the borrower and the book, etc.

A good scenario compares quite well to a traditional system test case and comprises (1) the functionality to be "tested", (2) actual test data (assumptions), and (3) expected result(s). A scenario for a library system could be the following:

> John Doe will borrow the book 1984; John Doe is a registered borrower and has never borrowed anything before; the book 1984 is available. After borrowing, John Doe will be registered as borrower of 1984 and 1984 will have a valid return date.

To fully validate a model, very many scenarios are needed. For example to test what happens when John Doe is not a registered borrower, has outstanding fines, or overdue books. What if the book 1984 is on loan, etc? It is important to define a limited number of scenarios that nevertheless cover all interesting situations.

Scenarios need not cover a complete system function, like the example above. It is meaningful to split long or complex scenarios into parts that can be handled separately. When role-playing complex scenarios, the parts that have been tested already can then be skipped over.

5.6 Prepare Scenario Role-Play

A CRC-team should consist of about 4–6 people with different backgrounds. During analysis it is for example useful to have domain experts, customer representatives or end users in the team. This helps to uncover missing requirements and helps team members to learn more about the domain, the actual system and its usage environment.

Before commencing the actual role-playing, the group members should make sure that they agree on all CRC-card descriptions. One team member should act as a scribe. The scribe records the role-play as it unfolds and helps the team to stay on the track of a scenario.

Each team member is responsible for a selection of CRC-cards assigned to him or her. This means that he or she is also responsible for all object cards that are instances of these CRC-cards and will therefore act out all objects corresponding to those cards.

Before a role-play is started an initial RPD is drawn that models the starting "state" of the scenario. All assumptions of the selected scenario must be reflected in this initial RPD. For the example scenario in Section 5.5, the initial RPD could look like the one in Figure 3. In this RPD, we have assumed two books and two borrowers. We have assumed a Librarian object that "knows" all books and all borrowers. We furthermore assume that there is some `userAgent` object that knows about the `:Librarian` object. The active object is marked by an asterisk.

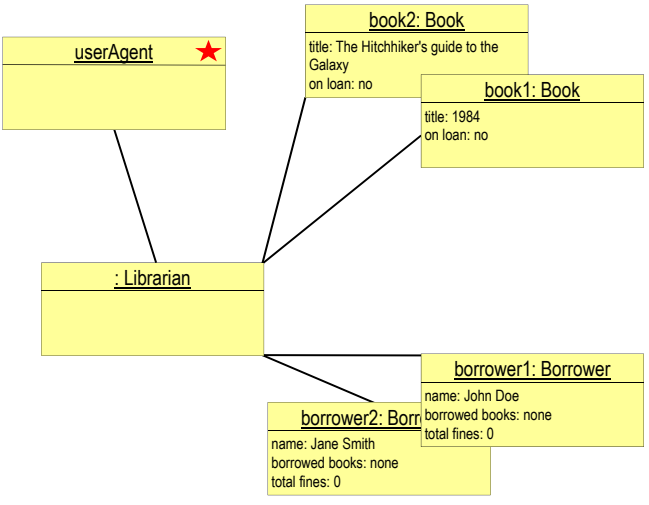


Fig. 3. Possible starting situation for scenario *John Doe will borrow the book 1984* with object `userAgent` marked as active

5.7 The Actual Role-Play

By means of role-playing scenarios the team kind of simulates how the future system would work, assumed it is build according to the CRC-card model.

Acting out a scenario is done by saying aloud what a certain object will do upon a request. All acting is restricted to the responsibilities noted on the CRC-cards and one must act only when it is one's "turn". The scribe can step in when user interaction or interaction with external systems is required.

During the first role-plays, many missing details will be uncovered and it will usually take a few scenarios before the model stabilises. It is strongly recommended to start with a very simple scenario for a "normal" case. Otherwise, there is a high risk of getting lost in discussions about large numbers of missing details.

What does *John Doe will borrow the book 1984* mean for our CRC-card model? How would this scenario translate into object responsibilities?

- Which object will start up the scenario? The corresponding object card in the RPD is marked as "active" and "control" is given to the holder of the corresponding CRC-card (the scribe in case of interface objects).
- What does the active object need to know to perform the requested work? Does it have the necessary responsibilities to fulfil the request? The active object says aloud all actions the active object would perform. Upon the request "are you on loan", the team member for the `book1` object might act as follows

  > OK, I should know whether I am on loan (since there is a corresponding responsibility on the `Book` CRC-card). Currently, I am not on loan (since the entry on the corresponding object card shows `on loan: no`).

After a request has been successfully handled, control goes back to the requester.

If responsibilities are missing on the corresponding CRC-card, the team discusses whether they should be added or if some alternative path should be tested. When CRC-cards are updated, their corresponding object cards are updated accordingly. If too much is missing to continue the role-play, the role-play is stopped.

– Are further collaborators needed to fulfil the request? Are their object cards present in the current RPD? Are their object cards connected to the current object's object card? Note that communication is only possible between connected object cards and to establish a connection with another object, it must know who the actual collaborator is.

  Object cards can be added to the RPD as necessary and connected to the objects that know about them. For dynamically created objects it is important to know which object is responsible for its creation. An object always knows the objects it created. Other objects must be somehow "introduced" to it. This can for example be done through a request carrying this information (a "parameter" as in request `4: borrow book1` in Figure 4). Collaborators can also be obtained as a result of a request. A search request for a book for example might return a specific book object.
– It is important to carefully walk through a scenario step by step. Sometimes however, it can be necessary to skip over certain details to be able to come to an end with a scenario. In such cases it is important to address these details later, for example by defining additional scenarios.

5.8 Record Scenarios

The scribe records scenarios as they unfold during the role-play. The scribe keeps track of all activities and updates the RPD accordingly. Recording scenarios in this way is very useful in several ways.

– During the role-play, the CRC team can easily check the current knowledge of the involved objects or whether two objects actually can communicate. This keeps the role-play on track and minimises the number of details the CRC team needs to memorise.
– Completed scenarios can be easily replayed after changes to the CRC model to verify whether they still work.
– They can be used to explain to others how the system works.
– It is easy to backtrack a few steps after a problem or after changing some decisions and resume the role-play from a consistent state.
– They can guide the implementation of the system.
– They can be used as system documentation.

Figure 4, shows the resulting RPD after completion of the scenario *John Doe will borrow the book 1984* from Section 5.5. It should be noted that the connecting line between `book1` and `borrower1` was not available at scenario start (see Figure 3).

It is drawn as an effect of request `4`, after the `:Librarian` object has identified the actual objects and can "tell" `borrower1` which actual book object it has to "talk to".

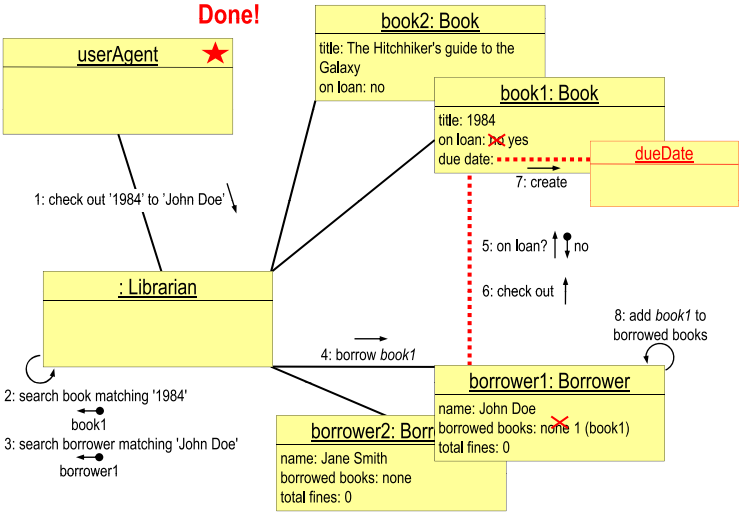


Fig. 4. RPD from Figure 3 after completion of scenario *John Doe will borrow the book 1984*

5.9 Update CRC-Cards and Scenarios

The scenario role-plays will likely uncover many problems in the CRC-card model. Whenever problems are detected, they need to be discussed and solved properly. All changes to the model must be made in a consistent way. Whenever a CRC-card is changed, scenarios and RPDs must be updated accordingly. Extensive changes might make it necessary to add scenarios covering the new or changed parts.

Missing responsibilities or collaborators can be added to suitable CRC-cards during the role-play. However, one should keep in mind that responsibilities should be distributed evenly and classes should have a clear single responsibility. Whenever there is no room left on a CRC-card, it should be considered to split it up.

6 The CRC/RPD-Process: Object-Oriented Design

The CRC/RPD approach can also be successfully applied in the design phase. The models from the analysis phase are refined and extended by addressing high-level implementation issues, like for example the actual organisation, management and storage of objects. One can also start considering programming language issues, like for example typing, visibility, packaging, or the usage of

library classes. Language dependencies should, however, be kept at a minimum to make the design as general as possible.

In the design phase, responsibilities will gradually be refined into actual attributes and methods. For the collaborators it has to be discussed how actual collaborations are established. This can be done by object creation, passing arguments, or receiving results.

In a good design classes should have single and well-defined purposes. As much information as possible should be hidden within the classes. Classes should communicate with as few as possible other classes and all communication should be explicit (no hidden collaborators). If communication must take place, as little as possible information should be exchanged. Following these principles minimises dependencies between classes and changes will propagate to few other classes, if at all.

The CRC/RPD approach can be applied for design exactly as described for analysis. The only difference is that more attention is given to solution domain issues. Evaluating designs in this way has many advantages.

- Models are easy to change; many design alternatives can be tested.
- Design decisions and the interactions between objects are well documented.
- It is less likely that implementation will run into problems caused by bad design decisions.
- The resulting designs will be less vulnerable to changes, since many alternatives have already been tested.

7 An Example Step-by-Step

In this section, we present a small case study for the application of our approach. We are modelling a library system for a university department, which is a slightly revised version of the CRC case study presented by Wilkinson [9].

> This application will support the operations of a technical library for a university department. This includes the searching for and lending of technical library materials, including books, videos, and technical journals. All library items have a registration code (research area code + running number).
>
> Each borrower can borrow up to 10 items. Each type of library item can be borrowed for a different period of time (books 6 weeks, journals 3 days, videos 1 week). If returned after their due date, the employee will be charged a fine, based on the type of item (books 5:-/day, journals and videos 20:-/day).
>
> Materials will be lent to employees only if they have (1) no overdue lendable items, (2) fewer than 10 articles out, and (3) total fines less than 100:-.

In the following subsections, we apply the CRC/RPD approach step by step as described in Sections 5.

7.1 Find Candidate Classes

Using the noun extraction approach, we underline all noun phrases in our problem description.

> This application will support the operations of a technical library for a university department. This includes the searching for and lending of technical library materials, including books, videos, and technical journals. All library items have a registration code (research area code + running number).
>
> Each borrower can borrow up to 10 items. Each type of library item can be borrowed for a different period of time (books 6 weeks, journals 3 days, videos 1 week). If returned after their due date, the employee will be charged a fine, based on the type of item (books 5:-/day, journals and videos 20:-/day).
>
> Materials will be lent to employees only if they have (1) no overdue lendables, (2) fewer than 10 articles out, and (3) total fines less than 100:-.

Discarding the most obvious duplicates and synonyms, we get the following initial list of candidates: `application`, `operations`, `technical library`, `university department`, `searching for`, `lending of`, `technical library materials`, `books`, `videos`, `technical journals`, `library items`, `registration code`, `research area code`, `running number`, `employee`, `type of library item`, `period of time`, `due date`, `borrower`, `total fines`, `users`, `overdue lendable items`, `articles`.

After further brainstorming, the CRC team might propose `librarian` as a further candidate and recognise further items that a library might need to handle, like for example `software` and `audio`.

7.2 Filter Candidates

It is important that the team members agree on a candidate's description and purpose. To start the discussion, the team member who proposed a candidate should explain his or her motives for doing so. In the following, we will briefly discuss all candidates from the list above.

- `Application`; refers to the system itself $\Longrightarrow$ Discard.
- `Operations`; irrelevant noise $\Longrightarrow$ Discard.
- `Technical library`; might also refer to the system itself. However, in this case it seems to refer to the stock or collection of things available for lending in the library. In a library system, we would need a class responsible for keeping track of the stock $\Longrightarrow$ keep as `Library`.
- `University department`; outside the problem domain $\Longrightarrow$ Discard.
- `Searching for` and `lending of`; verbs used as nouns $\Longrightarrow$ responsibilities $\Longrightarrow$ Discard.
- `Technical library materials`; the stock of the library. Important, but we have already chosen `Library` as responsible for that $\Longrightarrow$ Discard.
- `Books`, `videos`, `technical journals`, `software`, and `audio`; the objects in the library that represent books, videos, etc. They seem to be very similar with respect to borrowing. However, the problem description lists two differences; overdue fines and lending. A discussion might also reveal further differences in responsibilities. Videos and audio objects, for example, might have knowledge about playing times and software object need knowledge about system requirements $\Longrightarrow$ keep them all, but focus on the most important one (`Book`) first.

- `Library items`; a general term for books, videos, etc. Useful for keeping to keep the responsibilities that are common for all library items irrespective of their actual type in one place ⟹ keep `Library Item` as superclass for `Book`, `Video`, etc.
- `Registration code`, `research area code`, and `running number`; these are needed in our system, but there are no specific responsibilities connected to them. They are just properties of `Library Item` objects. This might however be different, if the system is responsible for assigning these codes in some automatic way. According to the current problem description, this is not the case ⟹ Discard.
- `Employee`; these are the borrowers in the library system. However, there are also borrower and users in the present list of candidates. From the problem statement it is not clear whether it is the responsibility of our system to check, if a borrower actually is an employee. For now, we assume that this responsibility is outside the scope of the present library system and the system only needs information about actual borrowers (whoever they are). In this case, borrower would be the best and most specific term to name a class responsible for borrower information and activities ⟹ keep `Borrower` and discard `Employee` and `User`.
- `Type of library item`; this information is already encoded in the different classes for the different types of library items ⟹ Discard.
- `Period of time`; the maximum time a certain type of library item can be borrowed. This is just a property of library items ⟹ Discard.
- `Due date`; this seems quite similar to `Period of time`. However, in contrast to `Period of time` there are actual responsibilities for due dates. Somehow, we need to check whether a due date has been passed. We also need a class responsible for the generation of due dates, when library items are borrowed ⟹ keep `Date`.
- `Fine`; there are different kinds of fines. Each type of library item has a potentially different fine per day to compute the fine for late returns. Borrowers have a total fine that must not exceed a certain amount. This amount is another kind of fine. However, all these fines are simple properties in different classes and do not have any specific responsibilities on their own ⟹ Discard.
- `Overdue lendable items`; knowing about overdue lendable items is a responsibility of borrowers. There are no specific responsibilities for overdue lendable items ⟹ Discard.
- `Articles`; used here as a synonym for library items ⟹ Discard.
- `Librarian`; this could be the class responsible for taking user requests, like checking in, checking out, and searching for library items. However, if `Librarian` is responsible for checking in, checking out, and searching, there is nothing left for `Library`, which already is responsible for the stock of library items. On the other hand, merging `Library` and `Librarian` into a single class might lead to a class with too many responsibilities ⟹ keep both `Librarian` and `Library`.

When reconsidering this list, `Lendable` is a much better class name than `Library Item`. This gives us the following list of candidates: `Lendable`, `Library`, `Librarian`, `Book`, `Video`, `Journal`, `Software`, `Audio`, `Borrower`, and `Date`.

7.3 Create CRC-Cards and Allocate Responsibilities

The responsibilities for `Lendable`, `Book`, `Video`, `Journal`, `Software`, and `Audio` are almost identical. To simplify the role-play, it is sufficient to start with one "representative" (`Book`). This will simplify our further activities considerably. Our initial set of CRC-cards will comprise six cards `Lendable`, `Library`, `Librarian`, `Book`, `Borrower`, and `Date` (see Figure 5).

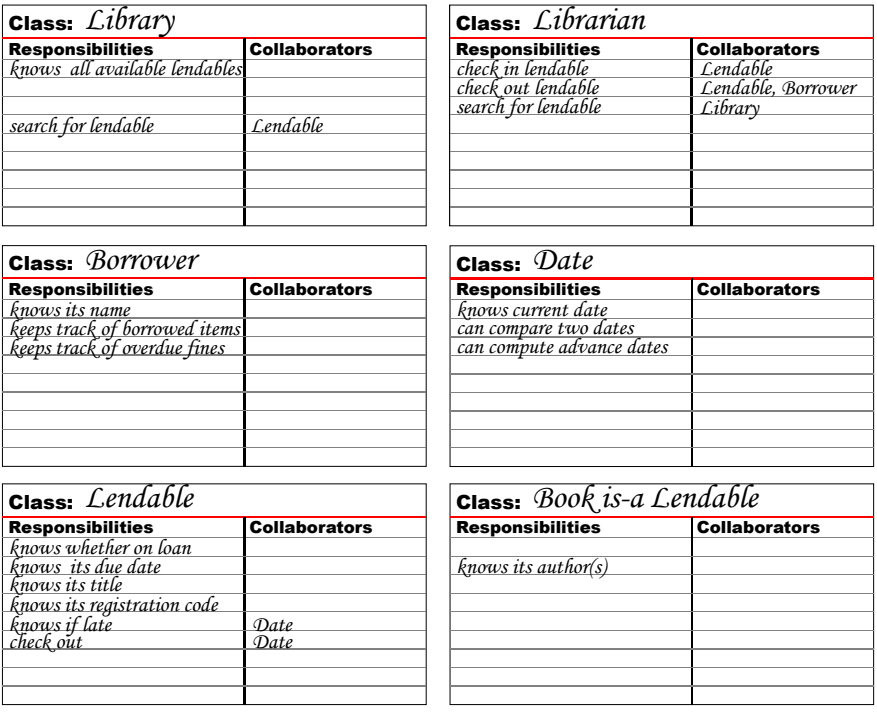


Fig. 5. CRC-cards for the Library Example

7.4 Define Scenarios

Scenarios are like test cases, we need a situation we want to test, specific test data to perform the actual test and a definition of the expected outcome. Without defining expected results, we cannot know whether the role-play actually was successful. As described in Section 5.5, we cannot start with a general scenario, like "what happens when an employee borrows a book". We would very likely get lost in making decisions about irrelevant details.

We will start with scenarios that cover the most basic and important functionality of the system, like checking out books. Furthermore, we take the non-problematic cases first to establish a stable working model. For the library example, we start with the following scenario:

> John Doe will borrow the book 1984. John Doe is a registered borrower in the system. Currently he has not borrowed any items and has no outstanding fines. The book 1984 is available and not on loan. After borrowing, John Doe will (still) be registered as borrower of 1984 and 1984 will have a valid return date.

Following this scenario, it is sensible to try some variations of this scenario for example where 1984 is on loan or John Doe has borrowed 10 books already (and therefore is not allowed to borrow more). Another important key scenario is returning books, like for example:

> John Doe returns the book 1984 on time. John Doe is a registered borrower in the system. Currently he has no outstanding fines. After borrowing, John Doe will no longer be registered as borrower of 1984. 1984 will no longer be on loan.

7.5 Prepare Group Session

A scribe is appointed and the six main CRC-cards are distributed among the other team members. Considering the scenario *John Doe will borrow the book 1984* from Section 7.4 and the CRC-cards in Figure 5, we define the initial RPD for our role-play (the starting "state").

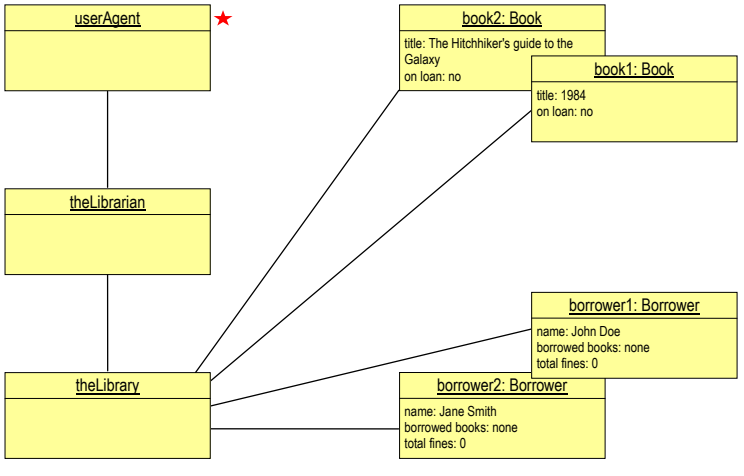


Fig. 6. Initial RPD for the scenario *John Doe will borrow the book 1984*

Close examination of our model reveals that there is no responsibility for keeping track of the borrowers. However, we have two classes that seem suitable for that; `Library` or `Librarian`. Since `Library` is already responsible for keeping track of all lendable items, it seems reasonable to add the responsibilities `knows all borrowers` and `search for borrower` (with collaborator `Borrower`) to the `Library` CRC-card.

For the role-play, it is sufficient to assume there are two books and two borrowers in the library. This results in the RPD shown in Figure 6. The currently active object is marked by a star. Please note that the `Book` objects do not know anything about the `Borrower` objects.

When role-playing a scenario, the following activities are carried out for each step in the scenario.

- Identify the currently active object and the current request it has to serve.
- If the request cannot be fulfilled directly, determine a reasonable (sub-) task to support the request.
  - Identify the object in the current RPD that has a responsibility matching this (sub-) task.

    If there is no such object, check whether there is a CRC-card with a matching responsibility. If so, evaluate why a corresponding object card is missing and create it eventually. If not, add the required responsibility to a suitable CRC-card and update the corresponding objects cards accordingly. Missing objects can indicate a problem with the model and/or scenario. Sometimes these problems cannot be solved easily and the model and/or scenario must be revised before the role-play can be continued.
  - Check whether the object in question is known to the currently active object. If not "announce" it somehow to enable communication (e.g. by providing a parameter).
  - The currently active object can now request the (sub-) task from another object and control is transferred to the object that provides the requested service;
  - The team member responsible for this object describes aloud (in first person) how the object provides this service.
- When the request is completed, control is transferred back to the requesting object.

7.6 Carry Out and Record the Actual Role-Play

The role-play starts at `userAgent` where we assume that someone has input the name of the borrower and the book title. We "translate" that into the first request `1: check out '1984' to 'John Doe'`.

In the remainder of this subsection, we present a transcript of a hypothetical but realistic role-play for the example scenario. The commentary illustrates problems that would typically be voiced by the scribe or another team member. Due to space limitations transcript is shortened slightly. The full transcript can be found in [5].

userAgent: I request from **theLibrarian** that the book with title "1984" is checked out to borrower "John Doe". I ask **theLibrarian**, since this is the only object I know.

Commentary: Control is transferred to **theLibrarian** object and the request is recorded in the RPD (see Figure 7).

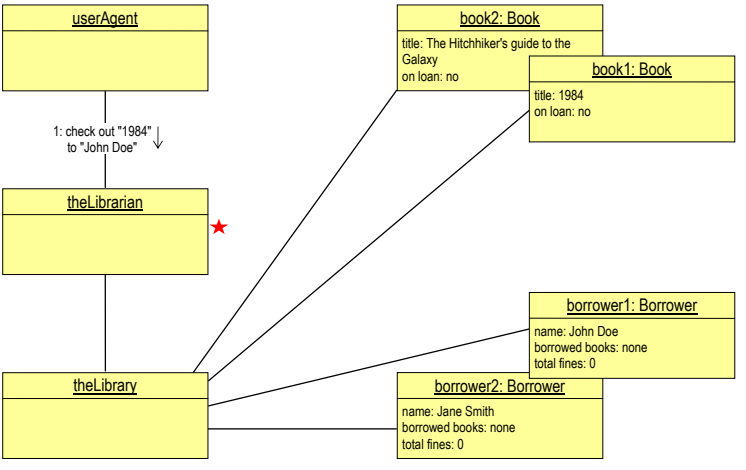


Fig. 7. RPD after the first step

theLibrarian: Fine, I actually have a responsibility **check out lendable**. To check out a book to a borrower, I have to check whether the book is available and whether the borrower is allowed to borrow. First, I ask the book whether it is on loan.

Commentary: How would you ask the book? You don't even know whether the book is in the library stock. Furthermore, you can only ask the particular book if you "know it" (i.e. are connected to it in the actual RPD).

theLibrarian continues: OK. I do not know anything about books in stock. My **check out lendable** responsibility has collaborators **Lendable** and **Borrower**. However, I cannot ask them, since I do not know the actual objects yet. I should probably ask **theLibrary**, since **Library** is the class that keeps track of the stock.

Commentary: The group discusses the problem. The **Library** CRC-card already has a suitable responsibility (**search for lendable**) and it is therefore reasonable to add **Library** as a collaborator to the **check out lendable** responsibility of the **Librarian** CRC-card. The **Librarian** CRC-card is updated and the role-play is resumed.

theLibrarian continues: I request from **theLibrary** object to **search for lendable** with the title "1984".

theLibrary: I can do that. I have responsibilities **knows all available lendables** and **search for lendable**. I go through all lendable items in stock and check if there is one with title "1984". I can do that, since I have **Lendable**

as a collaborator for `search for lendable` and each `Lendable` object `knows its title`. When I'm done, I know that `book1` is the `Book` object you are looking for.

Commentary: Now `theLibrary` has "introduced" `book1` to `theLibrarian`, which can now communicate with `book1` directly. The scribe draws a line between object cards `book1` and `theLibrarian` (see Figure 8). Control goes back to the requester (`theLibrarian`). Please note that we did not discuss in detail how the search is actually carried out. It is sufficient to know that it could easily be done by comparing titles of all known lendable items. We have also made a small "optimization" and did not explicitly ask all lendable items for their titles.

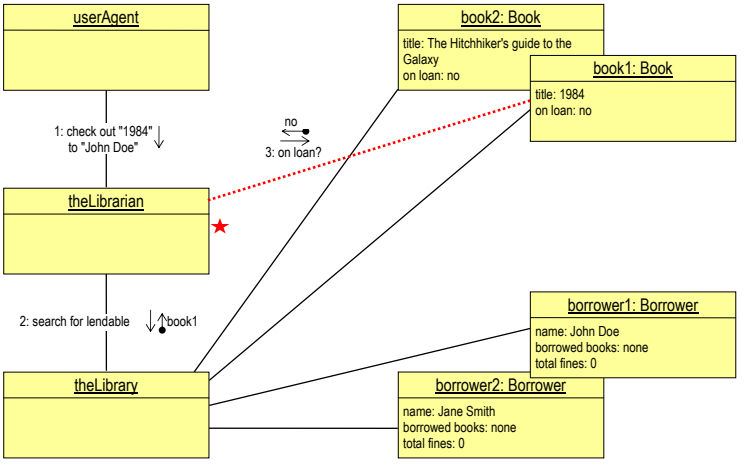


Fig. 8. RPD after `book1` has "answered" `no` on the request `on loan` from `theLibrarian`

`theLibrarian:` Now I can ask `book1` whether it is available.

`book1:` I have a responsibility `knows whether on loan` and my object card currently says `no`, i.e. I'm not on loan.

`theLibrarian:` Now I can continue with the check out. I also need to check whether "John Doe" is allowed to borrow. As with the book, we first have to get the actual borrower object. I request from `theLibrary` object to get me the borrower with the name "John Doe".

Commentary: This would work exactly as for looking up "1984" and we'll skip over the details.

`theLibrarian:` Now I need to check whether `borrower1` is allowed to borrow. However, I don't know how. I do not know anything about the rules for borrowing. Therefore, I ask `borrower1` directly; are you allowed to borrow?

`borrower1:` I don't know. I know my borrowed items and my overdue fines, but I don't know the rules for borrowing.

Commentary: The CRC team interrupts the role-play to discuss that problem. Who should have the responsibility to know whether a borrower is allowed to borrow? In the real world, this would be the librarian. However, in the software world the

borrower itself keeps track of its borrowed items and overdue fines[3]. A borrower could therefore easily check whether it is allowed to borrow without any need to collaborate with other objects. We add the responsibility `checks whether allowed to borrow` to the `Borrower` CRC-card. The role-play is resumed.

`borrower1` **continues:** I still do not know the rules for borrowing.

Commentary: The team decides to add another responsibility `knows rules for borrowing`. How this is actually realised is an implementation detail that will be discussed in the design phase. The role-play is resumed.

`borrower1` **continues:** OK. My object card says that I have not borrowed any books and that I do not have any overdue fines. Therefore, I can answer yes, I am allowed to borrow.

`theLibrarian`**:** Now we can finally check out `book1` to `borrower1`. First, I request `book1` to check itself out.

`book1`**:** I actually have a responsibility `check out`. When I check out myself, I must also take care of my responsibility `knows due date`. However, I do not know how to compute dates. I do not even know how long I can be borrowed.

Commentary: The rules for computing return dates differ for different types of `Lendable`. It is therefore reasonable to let the different types of `Lendable` be responsible for that. However, since all lendable items must have this responsibility, we add `knows maximum borrowing time` to `Lendable`. The role-play is resumed.

`book1` **continues:** I request `Date` to create a new return date 6 weeks from now.

Commentary: To create new objects it is sufficient to have the corresponding class as a collaborator. We can therefore create a new Date object and connect it to `book1` (see step 7 in Figure 9).

`Date`**:** I can do that. I have responsibilities `knows current date` and `compute new dates`.

Commentary: The scribe adds a new object card `returnDate` to the RPD and connects it to `book1`. Property `on loan` on `book1`'s object card can now be changed to `yes`.

`theLibrarian`**:** Now I can finally request `borrower1` to remember that it has borrowed `book1`.

Commentary: Now, we can draw a line between `borrower1` and `book1`, since `theLibrarian` has "announced" `book1` to `borrower1`. Since there are no responsibilities for adding and deleting lendable items, we must first add those to the Borrower card.

`borrower1`**:** Now I can add `book1` to my borrowed items.

`theLibrarian`**:** Now I'm done.

`userAgent`**:** I'm done too.

As pointed out earlier, there is no single correct way to model a system. In the role-play above, `theLibrarian` is the most active part, since it "executes" most of its `check out` responsibility on its own. However, this could have been done in many different ways. For example, `theLibrarian` could have delegated more

[3] Please note that a borrower cannot cheat in the software world. In the real world the librarian keeps own records about all borrowers' status. In the software world, these records are the actual borrower objects. There is no need for double bookkeeping.

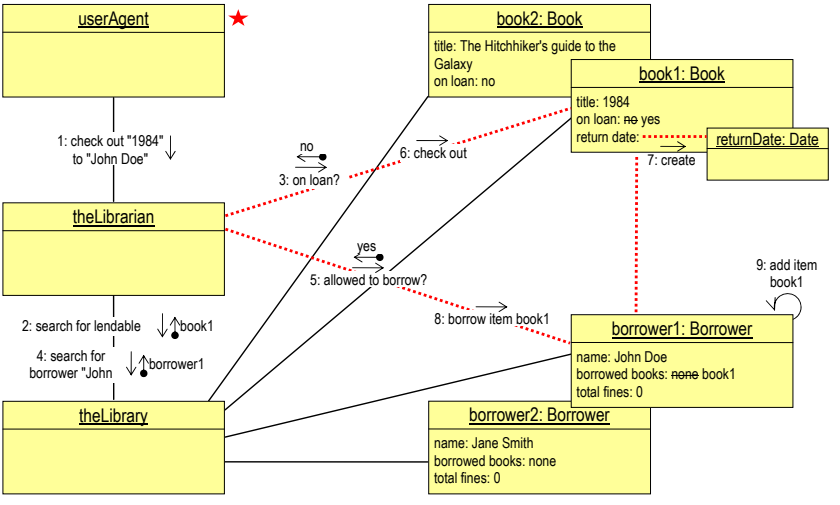


Fig. 9. Final RPD for the scenario *John Doe will borrow the book 1984*

sub-responsibilities to `borrower1`. The only necessary change to our earlier CRC-model would be to add a responsibility `borrow item` with collaborator `Lendable` to the `Book` card. The final RPD for this new model is shown in Figure 10.

This alternative way to model the system would have several advantages. The model would be less `Librarian`-centric and the responsibilities would be distributed more evenly throughout the system. On the other hand, this alternative seems less "natural" and therefore more difficult to understand.

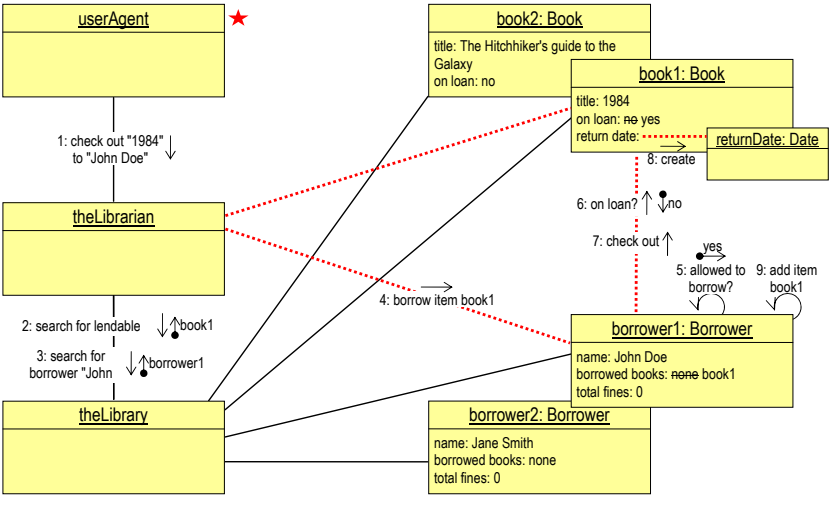


Fig. 10. Final RPD for the scenario *John Doe will borrow the book 1984 ...* for an alternative distribution of responsibilities

8 Possible Extensions or Improvements

Currently, we use simple connecting lines to express that two objects "know" each other. Adding semantic details to these associations (for example using UML syntax) could easily be done. However, we refused to do so to minimize the complexity of the diagrams. If more detailed information is needed we suggest using the RPDs as a starting point for documenting scenarios in full-scale UML interaction diagrams.

We have tried certain "syntactic sugar" to make the diagrams more expressive. Colour can for example be used to support the distinction between a scenario's starting state and the information that is added or changed during the role-play. The numbering scheme can for example be extended to also include the changes to object properties. That makes it easier to identify the changes that correspond to a certain requests. We leave such improvements to the CRC-teams, who are free to adapt the notation to their current needs. It is important to keep the diagrams as simple and flexible as possible (i.e. in the spirit of the CRC-cards themselves). The diagrams must support and not hinder the role-play activities.

9 Experience

We have started introducing CRC-cards in our introductory programming courses 2002 and added RPDs 2004. Our experience so far is quite promising. Teachers and TAs report fewer student problems with role-play activities, since we introduced RPDs.

Incorporating RPDs into the overall approach is no problem. The actual role-playing works exactly as without RPDs. The CRC-cards are distributed and the group members act out the scenario. The difference is that we now make a clear difference between classes and instances; CRC-cards represent classes and object cards represent their instances. The RPDs help students to actually record scenarios as they evolve. This makes them an excellent tool for consistency control and recovery. In addition they provide an excellent tool for the documentation of scenarios.

We have also analysed data from the student evaluations of all our introductory programming course offerings between fall 2001 and spring 2005. The syllabus during this time has been quite stable with one major exception. Since spring 2003, we no longer teach GUIs in the first course. A compilation of results can be found in Table 1. We have grouped the 12 offerings into four overlapping groups; courses covering neither CRC-cards nor RPDs, but including GUIs (column `GUI`, 2 offerings); courses covering either CRC-cards or RPDs (column `CRC/RPD`, 9 offerings); courses covering "pure" CRC-cards (column `CRC`, 6 offerings); and courses using RPDs to complement CRC-cards (column `RPD`, 3 offerings). One course teaching GUIs and CRC-cards was dropped from the compilation, since it did not fit any of the categories. More details about our experience can be found in [24].

Table 1. Student evaluations for 11 (of 12) courses spring 2001–spring 2005

Course category	GUI	CRC/RPD	CRC	RPD
sample size	126	267	156	111
course quality	4.04	3.34	3.26	3.51
student effort	3.52	3.60	3.46	3.89
positive/negative CRC/RPD	n.a.	2.55	2.78	2.25
%-age of students reporting problems	55.61	54.23	61.91	38.87

In Table 1, `sample size` is the number of students who submitted an evaluation form. On average 45.34% of the registered students of a course submitted a form. `Course quality` and `student effort` are measured on a Likert scale (1..5), where 1 means very low, 3 means average, and 5 means very high. `Positive/negative CRC/RPD` is the number of students clearly positive divided by the students clearly negative towards using CRC and/or RPDs[4].

As we can see from table 1, the %-age of students reporting problems has dropped since we introduced RPDs. Introducing CRC-cards and even more introducing RPDs also had an impact on the passing rates of our introductory programming courses. Figure 11 shows that passing rates largely followed the number of students applying for each seat in our Computer Science programs, except for two years; 2001, when we introduced CRC-cards and 2004, when we introduced RPDs. Since 2004, it furthermore seems that we could sustain our improved passing rates.

We have also taught several industry courses using this approach. However, we do not have actual data about its usage in industry.

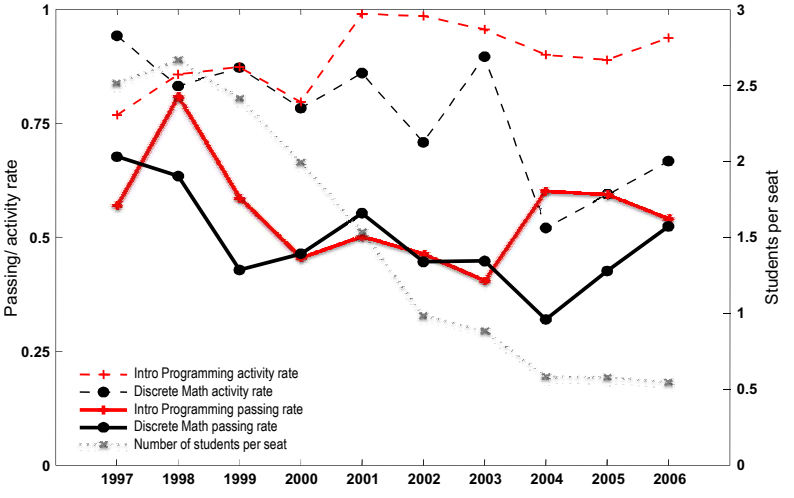


Fig. 11. Key data for the introductory programming course 1997–2006

[4] The actual question was “Have the CRC exercises been rewarding"?

10 Summary

CRC-cards are a simple yet powerful tool for collaborative object-oriented modelling. They help to explore a problem space and to better understand the problem at hand. Using scenario role-play, different object-oriented models can be evaluated in an early stage of software development. The approach described here gives a good understanding of the objects, responsibilities, and interactions involved in a solution for the problem.

The main advantages of the approach can be summarised as follows.

- The approach is low-tech and independent of programming languages. This makes it well suited for collaborative modelling in teams with people with very different backgrounds.
- The CRC-cards and RPDs form an excellent documentation of system analysis and initial design of a system. Using the RPDs one can easily see how the classes are supposed to work.
- Through scenario role-play, one can easily and systematically test alternative models using different sets of CRC-cards and varying responsibilities. This supports meaningful testing long before any code needs to be written.
- CRC-cards have shown to be useful in educational as well as in professional settings.
- Supporting the role-playing with RPDs has shown to be useful in educational settings.

References

1. Beck, K., Cunningham, W.: A laboratory for teaching object-oriented thinking. In: Proceedings OOPSLA 1989, pp. 1–6 (1989)
2. Bellin, D., Simone, S.S.: The CRC Card Book. Addison-Wesley, Reading (1997)
3. Biddle, R., Noble, J., Tempero, E.: Reflections on CRC cards and OO design. In: Proceedings Tools Pacific 2002, pp. 201–205 (2002)
4. Börstler, J., Johansson, T., Nordström, M.: Introducing OO concepts with CRC cards and Bluej—a case study. In: Proceedings FIE 2002, pp. T2G–1–T2G–6 (2002)
5. Börstler, J.: Object-oriented analysis and design through scenario role-play. Technical Report UMINF-04.04, Dept. of Computing Science, Umeå University, Umeå, Sweden (2004)
6. Coplien, J.: Experience with CRC cards in AT&T. C++ Report 3(8), 1,4–6 (1991)
7. Gray, K.A., Guzdial, M., Rugaber, S.: Extending CRC cards into a complete design process. Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA (2002), http://www.cc.gatech.edu/ectropic/papers/
8. Schulte, C., Magenheim, J., Niere, J., Schäfer, W.: Thinking in objects and their collaboration: Introducing object-oriented technology. Computer Science Education 13(4), 269–288 (2003)
9. Wilkinson, N.: Using CRC Cards, An Informal Approach to Object-Oriented Development. SIGS, New York, NY (1995)
10. Wirfs-Brock, R., McKean, A.: Object Design–Roles, Responsibilities, and Collaborations. Addison-Wesley, Boston (2003)

11. Wirfs-Brock, R., Wilkerson, B., Wiener, L.: Designing Object-Oriented Software. Prentice-Hall, Englewood Cliffs (1990)
12. Michalik, B., Nawrocki, J., Ochodek, M.: 3-step knowledge transition: A case study on architecture evaluation. In: Proceedings of the 30th International Conference on Software Engineering, pp. 741–748 (2008)
13. Haug, A., Hvam, L.: CRC cards to support the development and maintenance of product configuration systems. International Journal of Mass Customisation 3(1), 38–57 (2009)
14. Hvam, L., Riis, J., Hansen, B.L.: CRC cards for product modelling. Computers in Industry 50(1), 57–70 (2003)
15. Urquhart, C.: Bridging information requirements and information needs assessment: Do scenarios and vignettes provide a link? Information Research 6(2), Paper 102 (2001)
16. Andrianoff, S.K., Levine, D.B.: Role playing in an object-oriented world. In: Proceedings SIGCSE 2002, pp. 121–125 (2002)
17. Go, K., Carroll, J.M.: The blind men and the elephant: Views of scenario-based system design. Interactions 11(6), 44–53 (2004)
18. Moody, D., Benczúr, A., Demetrovics, J., Gottlob, G.: Cognitive load effects on end user understanding of conceptual models: An experimental analysis
19. Beck, K.: CRC: Finding objects the easy way. Object Magazine 3(4), 42–44 (1993)
20. West, D.: Object Thinking. Microsoft Press, Redmond (2004)
21. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1999)
22. Booch, G.: Object-Oriented Analysis and Design with Applications, 2nd edn. Addison-Wesley, Reading (1994)
23. Riel, A.J.: Object-Oriented Design Heuristics. Addison-Wesley, Reading (1996)
24. Börstler, J.: Improving CRC-card role-play with role-play diagrams. In: OOPSLA 2005 Conference Companion, pp. 356–364 (2005)

Reverse Engineering Using Graph Queries

Jürgen Ebert and Daniel Bildhauer

University of Koblenz-Landau
{ebert,dbildh}@uni-koblenz.de

Abstract. Software Reverse Engineering is the process of extracting (usually more abstract) information from software artifacts. Graph-based engineering tools work on fact repositories that keep all artifacts as graphs. Hence, information extraction can be viewed as querying this repository. This paper describes the graph query language GReQL and its use in reverse engineering tools.

GReQL is an expression language based on set theory and predicate logics including regular path expressions (RPEs) as first class values. The GReQL evaluator is described in some detail with an emphasis on the efficient evaluation of RPEs for reachability and path-finding queries. Applications for reverse engineering Java software are added as sample use cases.

1 Introduction

In software engineering, *modeling* is the process of abstracting parts of reality into a representation that abstracts from unnecessary details and allows automatic processing. Models may be either *descriptive*, if they represent existing artifacts, or *prescriptive*, if they are used as a blueprint for artifacts to be constructed.

In the domain of software engineering tools, a *modeling approach for software engineering artifacts* has to be chosen. The earliest tools for handling software artifacts were compilers which extracted tree models from program artifacts by a combined procedure of scanning and parsing, leading to *abstract syntax trees* as program models. Trees are easily representable in main memory, and there are many efficient algorithms for handling trees.

In general, trees are not powerful enough to keep all the necessary information about software engineering artifacts in an integrated form. As an example, definition-use chains or other additional links between the vertices of a syntax tree lead to more general graph-like structures. As a consequence of these shortcomings of trees Manfred Nagl [29] proposed to use *graphs* to represent source code in compilers and beyond that for all other artifacts in software engineering environments.

In this paper, the *TGraph approach* to graph-based modeling [15] is used to model software artifacts in software engineering tools. The generic graph query language GReQL is introduced which supports information extraction

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 335–362, 2010.

from graph repositories. An overview on GReQL is given and its central feature, the regular path expressions, is described in more detail. The focus is on the evaluation of path expressions to derive reachability and path information.

Section 2 describes the use of TGraphs as software models, Section 3 summarizes the necessary definitions on graphs and regular expressions, and Section 4 introduces the query language GReQL. In Section 5 the range of regular path expressions supported by GReQL is introduced, and the algorithms for evaluating these expressions are described in detail. Section 6 gives some examples of GReQL queries in reverse engineering, Section 7 discusses related work including some performance data, and Section 8 concludes the article.

2 Software

A *software system* does not only consist of its source code, but also of all other artifacts that are constructed and deployed during its development and its usage. These additional artifacts comprise a wide range of documents from requirements specifications, design and architecture models, to test cases and installation scripts. These artifacts are written in many different languages, some of them being of textual form others being visual diagrams. Some of them have a formal semantics others remain completely informal.

Graph Representation. To treat such a heterogeneous set of artifacts simultaneously in one tool environment, a *common technological space* is needed which is equally well-suited for storing, analyzing, manipulating and rendering them. Furthermore, it should be possible to also handle inter-artifact links, such as traceability information.

Regarding the work of Manfred Nagl and others, *graphs* form a good basis for such a technological space, since they are simultaneously able to model structure in a generic way and to also keep application-specific additional knowledge in supplementary properties like e.g. attributes.

Generally, every relevant *entity* in a software artifact can be modeled by a representative vertex that represents this entity inside a graph. Then, every (binary) *relationship* between entities can be modeled by an edge that carries all information relevant for this relationship. Note that the occurrence of an object o in some context c is a relationship in this sense, whereas o and c themselves are entities. Thus, occurrences of software entities in artifacts can be modeled by edges between respective vertices.

Example. As an example for the representation of a software artifact as a graph, the following listing shows a simple Java data structure for binary trees. Using a parser, this code can be transformed into an *abstract syntax graph (ASG)*, which consists of about 75 vertices and 100 edges for the code shown, if a fine-granular model is built.

Listing 1.1. "Java implementation of a binary tree"

```
1  class Node {
2    String data;
3    Node left, right;
4
5    public Node(String s) { data = s; }
6
7    public void add(String s) {
8      if (s.compareTo(data) < 0) {
9        if (left != null) left.add(s);
10       else left = new Node(s);
11     } else ...
12   }
13 }
```

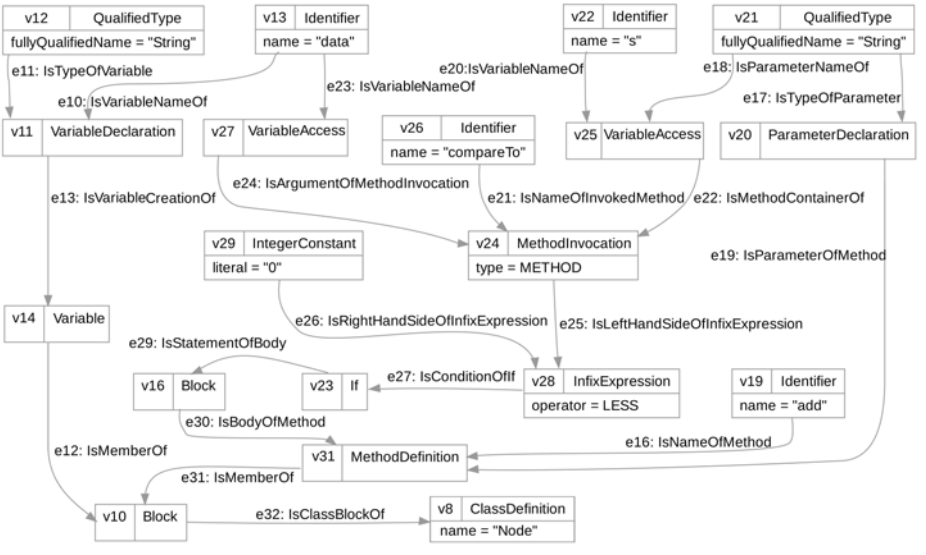


Fig. 1. Extract of the ASG for the class Node from listing 1.1

For brevity reasons, Figure 1 shows only a part of this graph, namely the definition of the class and its `data` attribute as well as the `add`-method with its outer `if` statement. The representation of the class `Node` itself is vertex `v8` at the bottom of the figure. It has the type `ClassDefinition` and holds the class name in its attribute called `name`. Methods and attributes of the class are grouped in a block represented by the vertex `v10`, which is connected to the class definition by the edge `e32`. In the same way, other elements of the source code are represented by vertices and edges connecting them. Analogically to vertex `v8`, all other vertices and edges are typed and may have attributes. For the sake of

clarity, the edge attributes keeping e.g. the position of the occurrences in the source code are omitted. Here, the ASG is a directed acyclic graph (and not only a tree). Every entity is represented only once but may occur several times (e.g. vertex `v13`).

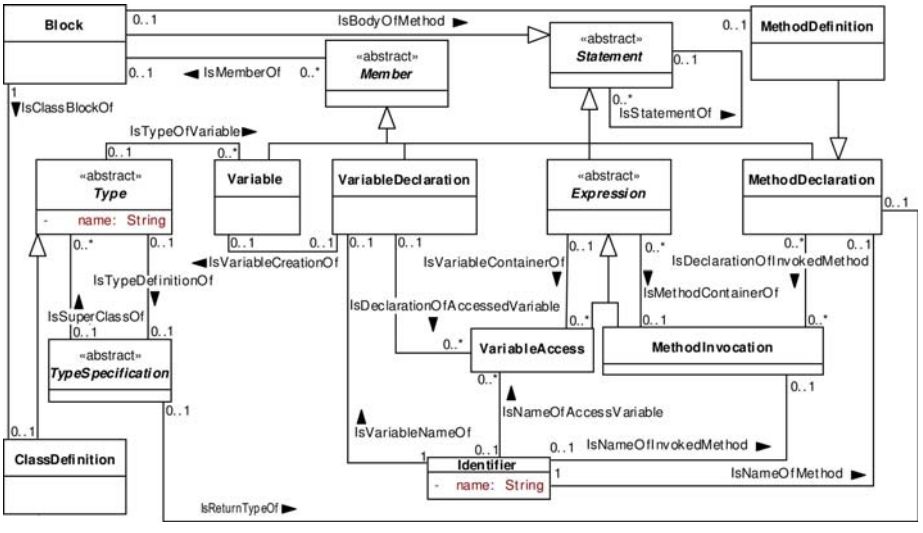


Fig. 2. Extract of the Graph Schema for the Java language

Graph schema. The types and names of the attributes depend on the types of the vertices and edges of the graph. These types can be described by a *graph schema* which acts as a metamodel of the graph. Since the Meta Object Facility (MOF) [30] is a widely accepted standard for metamodeling, Figure 2 shows a small part of a metamodel for ASGs representing Java programs depicted as a MOF compatible UML class diagram. The classes denote vertex types while the associations between classes denote edge types. Thus, class diagrams may be used to describe graph schemas.

The vertex type `ClassDefinition` mentioned above is depicted at the left border of the diagram. It is a specialization of the vertex type `Type` and inherits the `name`-attribute from this class. Every class consists of a `Block`, which is connected to the class by a `IsClassBlockOf` edge as it was depicted in the example graph above. A `Block` is a special kind of a `Statement` and groups other `Statements`. The edge type `IsStatementOf` represents the general occurrence of one statement in another and is the generalization of all other edge types which group statements in each other. These generalizations are omitted in the diagram for reasons of brevity but are important in the queries shown later in this paper.

Graph-based Tools. Schemas can be defined for all kinds of languages. There are schemas for textual programming languages, and there are schemas for visual languages. Schemas can be derived on all levels of granularity from fine-granulare abstract syntax over middle level abstractions to coarse architecture descriptions depending on their purpose. Different schemas can also be combined to larger integrated schemas. Metamodel Engineering is the discipline that deals with topic.

Given a graph schema, tools can work on graphs conforming to it. These tools extract the relevant facts from software engineering artifacts (e.g. by parsing) and store them as graphs in a graph repository. Several services on the repository (like enrichment, analysis, abstraction, and transformation) help to use these graphs for solving software engineering problems. These services may be application specific or generic.

One of the services to gather information from the repository is *querying*. In the following, this paper focuses on the generic query language GReQL. GReQL works on TGraphs, which are introduced in the following section.

3 Terminology

To describe the graph-based approach to build reengineering tools in more detail, an appropriate terminology is needed. This section introduces TGraphs and a suitable notation for algorithms on TGraphs. Furthermore, some facts about regular languages are enumerated, which are needed later for the treatment of regular path expressions.

3.1 Graphs

To establish a comprehensive graph technology on a formal basis, a precise definition of its underlying concepts is essential. In this paper, TGraphs are used.

TGraphs. TGraphs are a powerful category of graphs which are able to model not only structural connections, but also all type and attribute information needed for an object-based view on the represented model. TGraphs are typed, attributed, and ordered directed graphs, i.e. all graph elements (vertices and edges) are typed and may carry type-dependent attribute values. Furthermore, there are orderings of the vertex and the edge sets of the graph and of the incidences at all vertices. Lastly, all edges are assumed to be directed.

Definition: TGraph

Let
- *Vertex be the universe of* vertices,
- *Edge be the universe of* edges,
- *TypeId be the universe of* type identifiers,
- *AttrId be the universe of* attribute identifiers, *and*
- *Value be the universe of* attribute values.

Assuming two finite sets,
- a vertex set $V \subseteq Vertex$ *and*
- an edge set $E \subseteq Edge$,

be given. $G = (Vseq, Eseq, \Lambda seq, type, value)$ *is a* TGraph *iff*
- $Vseq \in \mathsf{iseq}\, V$ *is a permutation of* V,
- $Eseq \in \mathsf{iseq} E$ *is a permutation of* E,
- $\Lambda seq : V \to \mathsf{iseq}(E \times \{in, out\})$ *is an* incidence function *where*
  $\forall\, e \in E : \exists! v, w \in V : (e, out) \in ran\, \Lambda seq(v) \wedge (e, in) \in ran\, \Lambda seq(w)$,
- $type : V \cup E \to TypeId$ *is a* type function, *and*
- $value : V \cup E \to (AttrId \twoheadrightarrow Value)$ *is an* attribute function *where*
  $\forall\, x, y \in V \cup E : type(x) = type(y) \Rightarrow dom(value(x)) = dom(value(y))$.

Thus, a TGraph consists of an ordered vertex set V and an ordered edge set E. They are connected by the incidence function Λseq which assigns the sequence of its incoming and outgoing edges to each vertex. For a given edge e, $\alpha(e)$ and $\omega(e)$ denote its *start vertex* and *target vertex*, respectively. Furthermore, all elements (i.e. vertices and edges) have a type and carry a type dependent set of attribute-value pairs. Figure 1 visualizes an example TGraph.

Further Graph Properties. Given a TGraph G, paths are used to describe how a given vertex w may be reached from a vertex v.

Definition: Path

A path *from* v_0 *to* v_k *in a TGraph* G *is an alternating sequence*
$C =< v_0, e_1, v_1, ..., e_k, v_k >, k \geq 0$,
of vertices and edges, where
$\forall\, i \in \mathbb{N}, 1 \leq i \leq k : \alpha(e_i) = v_{i-1} \wedge \omega(e_i) = v_i$.
v_0 *is called the* start vertex $\alpha(C)$ *and* v_k *the* target vertex $\omega(C)$ *of the path. A path is called a* proper path, *if all its vertices are distinct.*

Definition: Derived Edge Type Sequence

Given a path C *the corresponding sequence of edge types*
$< type(e_1), type(e_2), ..., type(e_k) >$
is called its derived edge type sequence.

The existence of an edge from v to w is denoted by $v \to w$, and the existence of a path by $v \to^* w$. Furthermore, $v \to^*$ denotes the set of all vertices reachable from v by any path.

3.2 Pseudocode

Given an appropriate data structure for TGraphs [12], graph algorithms can be implemented in a such a way that graph traversals are efficient. There is a Java-API called JGraLab[1] that allows a convenient and concise notation of algorithms which is very near to pseudo code.

[1] `http://www.ohloh.net/p/jgralab`

The pseudo code used in this article adheres to JGraLab's conventions. There is a type `Graph`, and graph elements are instances of the types `Vertex` and `Edge`, respectively. A few of the API operations are listed here:

```
interface Vertex {
  /** @return the sequence of outgoing edges at this vertex */
  Iterable<Edge> getAllOutEdges ();
  ...
}
```

```
interface Edge {
  /** @return the start vertex of this edge */
  Vertex getAlpha ();

  /** @return the end vertex of this edge */
  Vertex getOmega ();
  ...
}
```

According to these interfaces, the edges incident to a vertex v can be traversed in the order defined by $\Lambda seq(v)$ using a `for`-loop like

```
for (Edge e: v.getAllOutEdges()) {
  // process edge e
}
```

3.3 Regular Expressions and Automata

Since regular expressions are in center of the query language described in this article, some basic facts about finite automata and regular expressions are compiled here ([20]).

Definition: Regular Expression

Given some alphabet Σ, the regular expressions (REs) *over Σ are defined inductively as follows:*
(1) Φ is a regular expression and denotes the empty set $\varnothing$.
(2) ϵ is a regular expression and denotes the set $\{\epsilon\}$.
(3) For each $a \in \Sigma$, a is a regular expression and denotes the set $\{a\}$.
(4) If r and s are regular expressions denoting the languages R and S, respectively, then concatenation (rs), choice $(r + s)$, and closure r^* are regular expressions that denote the sets RS, $R \cup S$, and R^*, respectively.
The set of strings denoted by a regular expression r is called $L(r)$, the language *described by r.*

Languages described by regular expressions can also be described by deterministic finite automata.

Definition: Deterministic Finite Automaton (DFA)

A deterministic finite automaton (DFA) $dfa = (S, \Sigma, \delta, s_0, F)$ *over* Σ *consists of*

- *a set* S *of states,*
- *a transition function* $\delta : S \times \Sigma \to S$*,*
- *a start state* s_0*, and*
- *a set* $F \subseteq S$ *of terminal states.*

DFAs can be visualized as graphs, where the vertices are the states of the automaton and where there are edges $e = s_1 \to_a s_2$ iff δ maps (s_1, a) to s_2. An example is shown in figure 4.

To use automata in software, they may be implemented as graphs, as well. Only a few methods on automata are used in this paper:

```
1  interface Automaton extends Graph{
2    /** @return the state state
3    Vertex getStartState ();
4
5    /** @return true iff s is a terminal state
6    boolean isTerminal (Vertex s);
7
8    /** @return the sequence of all enabled transitions of type t out of state s */
9    Iterable<Edge> getAllEnabledTransitions (Vertex s, Type t);
10   ...
11 }
```

An automaton *dfa* accepts a string l over an alphabet Σ by a state $s \in S$ iff l is the derived type sequence of a path from s_0 to s in the graph of *dfa*. The set of strings accepted by s is called $L(dfa, s)$. Consequently, the *language* accepted by an automaton *dfa* is $L(dfa) := \cup_{s \in F} L(dfa, s)$, where F is the set of terminal states.

It is well known that every language accepted by some finite automaton is also describable by a regular expression and vice versa [20]. It is possible to construct an equivalent automaton $dfa(r)$ from a given regular expression r using Thompson's construction [35] followed by Myhill's algorithm [28].

Though in theory the size of the deterministic finite automaton built from a given regular expression r of size n can be of the order 2^n, this 'explosion' does not occur frequently in practice [1]. Experience shows that DFA-acceptors for regular languages have usually only about twice the size of their regular expressions.

4 Querying

Since graphs are subject to algorithms, all decidable problems on graphs can in principle be solved by respective graph algorithms. This approach affords a lot of effort on the user side, since each request for information about the model represented by a graph leads to a specific problem which has to be solved

algorithmically. Much of this work can be avoided in practice by supplying the users with a powerful *querying facility* that allows easy end user retrieval for a large class of information needs.

GReQL. In the case of TGraphs, the *Graph Repository Query Language GReQL* supplies such a powerful facility for end user querying. GReQL was developed in the GUPRO project [14] at the University of Koblenz-Landau. The current version GReQL 2 was defined by Marchewka [25]. There is a full optimizing evaluator for GReQL available on JGraLab implemented by Bildhauer and Horn [7,21].

GReQL can be characterized as a schema-sensitive expression language with dynamic types. It gives access to all graph properties and to the schema information. GReQL queries may be parameterized. Its most valuable feature is an elaborated concept for path expressions.

Since GReQL is an expression language, its *semantics* is self-evident. Every GReQL query is a GReQL expression e which is either atomic or recurs to some partial expressions $e_1, ..., e_k$ whose evaluation is used to compose the resulting value of e. Thus, using mathematical expressions as building blocks (see below) the value of a GReQL expression is precisely defined.

The type-correctness of GReQL expressions is checked at evaluation time. The type system itself is encapsulated in a composite Java class called `JValue` (see below) in the context of JGraLab.

Querying. The goal of *querying* is to extract information from graph-based models and to make the extracted information available to clients, such as software engineers or software engineering tools.

Querying is a *service* on TGraphs that - given a graph g and a GReQL query text q - delivers a (potentially composite) object, that represents the query's result. The universe of possible query results is given by `JValue`.

Specification: Querying

Signature:
$query : TGraph \times String \nrightarrow JValue$

Pattern:
$s = query(g, q)$

Input:
a TGraph g and a valid GReQL query q

Output:
a JValue s which contains the result of evaluating q on g (according to the semantics of GReQL)

Types. The types of GReQL values are defined by the Java class `JValue`, since the values delivered by GReQL queries are usually passed on to Java software, e.g. for rendering the result. `JValue` is a union type which comprises all possible types

that a GReQL result may have. Its structure is that of a composite, i.e. there are some basic (atomic) types and some composite types that allow structured values.

Basic types are `integer`, `double`, `boolean`, and `string`, and also a concept for enumeration types is supplied. Some graph constituents are also supported as types, namely `vertex`, `edge` and `type`, the former referring to vertex or edge types in the schema.

Composite types allow structured data (tuples, lists, sets, bags, tables, maps, and records). Also paths, path-systems (see below) and graphs are supported as types as well as path expressions.

Expressions. Since GReQL is an *expression language* the whole language can be explained and implemented along its different kinds of expressions:

```
1  0, 123                          // integer literals
2  0.0, -2.1e23                    // double literals
3  true, false                     // boolean literals
4  "hugo", "ab\n"                  // string literals
5  v                               // variable expression
6  let x := 3 in x + y             // let-expression
7  x + y where x := 3              // where-expression
8  sqr(5)                          // function application
9  not true                        // unary operator expression
10 b and c, x > 0                  // binary operator expressions
11 v.name                          // value access
12 x > 5 ? 7 : 9                   // conditional expression
13
14 // quantified expressions
15 exists v:V{MethodDeclaration} @ outDegree(v)=0
16 forall e:E{IsCalledByMethod} @ alpha(e) = omega(e)
17
18 // FWR expression
19 from caller, callee:V{MethodDeclaration}                  //(1)
20 with caller <--{IsBodyOfMethod} <--{IsStatementOf}*      //(2)
21            <--{IsDeclarationOfInvokedMethod} callee
22 report caller, callee end                                //(3)
```

For all basic types (`integer`, `double`, `boolean`, `string`, `enum`) appropriate notations for literals are defined (lines 1 to 4). Variables stand for the value which is bound to them (line 5). Since GReQL is a *single assignment language*, variables may be bound only once in a given scope. All attribute and type identifiers from the schema are predefined variables. Local scopes may be formed using `let`- or `where`-constructs (lines 6 and 7).

Composite expressions may be constructed using function applications (line 8) or using unary or binary operators (lines 9 to 10), including a conditional clause . (Besides the usual operators known for the basic types, there are also

special operators, called regular path descriptions (RPEs) which are themselves expressions. RPEs and their usage are described in more detail in Section 5.) Attributes of graph elements are accessed using a dot notation (line 11).

Since GReQL is heavily based on set-like expressions, also quantified expressions can be used using `exists-` and `forall`-quantifiers (lines 15 to 16). Here the restriction holds that the domains of the bound variables have to be finite.

The last form of expressions supplied by GReQL is the from-with-report(FWR) expression (lines 19 to 22). Its result is a bag of values (or of tuples of values, depending on the length of the report list) containing the results of the expressions in the report clause (line 22) evaluated for each variable binding in the declaration (line 19) which fulfills the condition of the with clause (lines 20 and 21) . Alternatively the result may also have the form of a table, a set or a map.

Function Library. The GReQL types go along with a set of operations that can be applied on their instances. Besides the usual standard operations on basic types, there is also a long list of operations on graph elements (like `degree()`, `alpha()`, `omega()`, etc.) and aggregation operations (like `avg()`, `count()`, etc.). These operations are kept in an editable function library which can be extended easily. At present it contains about 100 functions.

Assuming that all functions in the library are polynomial in the size of the graph, all GReQL expressions can be evaluated in polynomial time, if all variables in quantified and FWR expressions are bound to sets whose size is also polynomial in the graph size.

5 Path Expressions

The language described up to now supports the evaluation of expressions in the usual domains of arithmetics, boolean values, and strings and thus gives a framework for extracting information from graph elements. But it does not yet give enough support for structure dependent information extraction.

Support for such a kind of information extraction is given by GReQL's so-called *regular path expressions*. To exploit the knowledge encoded in the structural part of a TGraph, connection patterns can be described by regular expressions over the set of element types.

5.1 Definitions

Simplified Syntax. Path expressions allow the comprehensive description of the sets of all edges that have the same derived edge type sequence. GReQL uses *regular* path expressions as a means to support navigation in queries.

Definition: Regular Path Expressions (simplified)

A regular path expression (rpe) *is a non-empty regular expression over the set* $EdgeTypeID \subseteq TypeID$ *of all edge types, according to the following rules*

(i) Given $t \in EdgeTypeID$, `-->{t}` *is an rpe.*

(ii) Given two rpes rpe_1 and rpe_2, $(rpe_1 rpe_2)$ is an rpe. [concatenation]
(iii) Given two rpes rpe_1 and rpe_2, $(rpe_1 \mid rpe_2)$ is an rpe. [choice]
(iv) Given an rpe rpe, (rpe^*) is an rpe. [closure]

According to subsection 3.3, a regular path expression *rpe* defines a language $L(rpe)$ over *EdgeTypeID*. Assuming the usual precedences (concatenation before choice before closure) unnecessary parentheses may be skipped.

This definition is simplified in the sense that only forward arrows `-->` are used which describe edges in their original direction. There are several other edge notations which may be used, as well:

`<--` describes an edge traversed in the opposite direction.
`<->` describes an edge traversed in any direction.
`<>--` describes an aggregation edge traversed from its aggregate's side.
`--<>` describes an aggregation edge traversed from its component's side.

Semantics of Path Expressions. The semantics of a regular path expression *rpe* is the set of all paths, whose derived edge type sequence conforms to the language defined by the *rpe*.

Path descriptions are used as abstract operators. Assuming that *rpe* is a regular path expression and v, w are vertices, there are several ways to apply *rpe*:

- `v rpe` is the set of vertices reachable from `v` according to `rpe`.
- `rpe w` is the set of vertices from which `w` is reachable according to `rpe`.
- `v rpe w` is the condition that `w` is reachable from `v` according to `rpe`.
- `path(v,rpe,w)` is a path from vertex `v` to vertex `w` if it exists.
- `pathSystem(v,rpe)` is a path system containing exactly one path for every vertex reachable from `v` according to `rpe`.
- `pathSystem(rpe,w)` is a path system containing exactly one path from every vertex from which `w` is reachable according to `rpe`.

All these applications can be evaluated by *search algorithms* on vertices which are explained in more detail in the next subsections.

Full Syntax. Besides the simplified definition above, there are several other notations which are also allowed in GReQL path expressions. They all can be handled as well inside the search algorithms to be described below.

(1) Restrictions to the vertex types of a path can be added by using an `&` sign in front of a type in braces, e.g. `&{MethodDeclaration}` states that only `MethodDeclaration`-vertices are allowed.
(2) For vertices and edges also boolean expressions are allowed instead of type restrictions, where the current element is denoted by `thisVertex` or `thisEdge` respectively.
(3) A specific edge can be attached to an edge symbol by embedding any expression that evaluates to an edge, as in `--e->` where `e` is a variable containing an edge. Similarly vertex variables like `v` can be used to denote a special vertex.

(4) Role names defined in the schema can be used instead of or additional to edge types. E.g. `<->{@member}` restricts the set of edges to those which are incident to a vertex with role `member`
(5) There are further operations on regular path expressions such as transitive closure (`rpe+`), exponentiation (`rpe^n`), option (`[rpe]`) and transposition (`rpe^T`).

Example. As an example, the query below denotes the set of all classes containing a method which calls a method of class `c` in a graph according to the schema of Figure 2.

```
1 c <--{IsClassBlockOf} <--{IsMemberOf} &{MethodDeclaration}
2   -->{IsDeclarationOfInvokedMethod} <--{IsMethodContainerOf}
3   -->{IsStatementOf}* -->{IsBodyOfMethod}
4   -->{IsMemberOf} -->{IsClassBlockOf} &{thisVertex <> c}
```

Here, all vertices reachable from a class c via paths according to the regular path expression are returned. These paths have the following structure: They lead from c to some member m of type `MethodDeclaration` (line 1). Starting from m, they find some call expression (line 2) where m is called. Then, the method $m1$ is determined that this call expression belongs to (line 3). Finally, the class of $m1$ is derived, and it is assured that this class is different from c.

5.2 Search Algorithms

Regular path expressions can be evaluated efficiently by *search algorithms* on the given graph. This holds for all features of regular path expressions described.

In the following, the algorithms needed for the simplified syntax are explained in detail. It should be obvious how these algorithms can be extended to handle the full syntax.

Search Algorithms. Search algorithms are *traversal algorithms*, i.e. they visit every graph element of (a part of) a graph exactly once. Search algorithms mark every vertex they visit to avoid double visits and use a collection object (usually called a *work list*) to control the spreading of this marking over the graph.

The set of marked vertices can be kept in a *set structure* according to the interface `Set`:

```
1  interface Set<E> {
2    /** @return true iff this set contains no elements */
3    boolean isEmpty ();
4
5    /** inserts the element x into this set */
6    void insert (E x);
7
8    /** @return true iff the element x is a member of this set */
9    boolean contains (E x);
10 }
```

The work list can be stored in any *list-like structure*, described by the interface `WorkList`:

```
1  interface WorkList<E> {
2    /** @return true iff this worklist contains no elements */
3    boolean isEmpty ();
4
5    /** inserts the element x into this worklist (possibly multiple times) */
6    void insert (E x);
7
8    /** returns (and deletes) an arbitrary element from the list */
9    E extractAny ();
10 }
```

Reachability. Given implementations for the interfaces `Set` and `WorkList`, a simple search algorithm can be given that visits all vertices in

$$reach_G(i) := \{v \in V \mid \exists\, C : Path \bullet \alpha(C) = i \wedge \omega(C) = v\}$$

i.e. the set of all vertices reachable from a given vertex i.

In the following pseudocode "visiting" a vertex v or an edge e is expressed by action points, which are noted as pseudo-comments like

`// process vertex v` or
`// process edge e`,

respectively. At these action points, potential visitor objects may be called to execute some action on the respective element:

```
1  Algorithm: SimpleSearch:
2  ~~~~~~~~~~~~~~~~~~~~~~~
3  // vertex−oriented search starting at vertex i
4
5  Set<Vertex> m = new ...;
6  WorkList<Vertex> b = new ...;
7
8  void reach(Vertex i) {
9    m.insert(i);
10   // process vertex i
11   b.insert(i);
12   while (! b.isEmpty()) {
13     // inv: set(b) ⊆ m ⊆ i →*
14     v = b.extractAny();
15     for (Edge e: v.getAllOutEdges ()) {
16       // process edge e
17       w = e.getOmega();
18       if (! m.contains(w)) {
19         // process tree edge e
20         m.insert(w);
21         // process vertex w
22         b.insert(w);
23 } } } }
```

During the `while`-loop the work list b invariantly contains only marked vertices that are reachable from i. This is expressed by the invariant in line 13, where $set(b)$ denotes the set of all elements contained in b. Thus, all marked vertices are reachable. Conversely, any vertex reachable from i will eventually be included into b. Since insertion into m is only done for non-marked vertices and a vertex inserted into b is marked simultaneously, no vertex is inserted twice into b.

Every vertex is extracted at most once from b and the inner `for`-loop traverses its outgoing edges only. Thus, the body of the inner loop is executed at most once for each edge. Together this leads to a time complexity of $\mathcal{O}(max(n, m))$ for a graph with n vertices and m edges.

Reachability Tree. It is well-known that the spreading of the marking over the graph defines a spanning tree of the marked part. This tree is rooted in i and contains an incoming edge for every other vertex reachable from i.

Such a tree can be represented by a predecessor function

$$\texttt{parentEdge} : V \nrightarrow E,$$

which assigns its incoming edge to each non-root vertex. Such a tree is called *reachability tree* for $i \rightarrow^*$.

Partial functions like `parentEdge` on the vertices can be stored in a map-like data structure, according to the interface `VertexMarker` which stores at most one value at every vertex:

```
1  interface VertexMarker<E> {
2    /** stores the value x at the vertex v */
3    void setValue (Vertex v, E x);
4
5    /** @return the value stored at the vertex v */
6    E getValue (Vertex v);
7  }
```

Given such a vertex marker, the computation of `parentEdge` can be done in the algorithm `SimpleSearch` by refining the action point where the tree edges are processed. Here, e is the current edge and w is the newly marked vertex:

```
1  VertexMarker<Edge> parentEdge = new ...;
2
3  process tree edge e:
4  ~~~~~~~~~~~~~~~~~
5    parentEdge.setValue(w,e);
```

Paths. Given `parentEdge`, as computed by the algorithm `Simple Search`, a corresponding path $i \rightarrow^* v$ is implicitly given for every vertex v reachable from i. Such a path can be issued in reverse order by *backtracing* over the tree edges starting at v.

```
process path i →* v:
~~~~~~~~~~~~~~~~~~
  z := v;
  //process path vertex z;
  while (z != i) do {
    e := parentEdge.getValue (z);
    // process path edge e;
    z := e.alpha();
    // process path vertex z;
  }
```

Shortest Paths. The work list used in search algorithms can be implemented in various ways. It is well-known, that a queue-like implementation leads to a *breadth-first search (BFS)* approach, whereas a stack-like implementation implies a *depth-first search (DFS)* of the graph.

The breadth-first implementation of graph traversal is particularly interesting, since it implies that the path-tree is a *shortest path-tree*, i.e. all paths in the path tree have a minimum number of edges. Thus, GReQL uses breadth-first-search for the evaluation of regular path expressions.

Example. Figure 3(a) contains the sketch of a TGraph to demonstrate the effect of Algorithm `SimpleSearch` and the computation of `parentEdge`. Assume that the vertex set $\{A, B, C, D, E\}$ and the edge set {1,2,3,4,5,6,7} as well as the incidences are ordered according to their identifiers. The edge types $\{a, b\}$ are also shown.

(a) TGraph (b) BFS Tree

Fig. 3. Sample TGraph and its BFS tree

Using a breadth-first search starting in vertex A, the tree shown in Figure 3(b) is derived. It shows that all vertices are reachable from A. E.g. a path for vertex E can be derived by backtracing from E:

$< A, 1, B, 5, D, 7, E >$.

5.3 Automaton-Driven Search Algorithms

Algorithm `SimpleSearch` explores parts of the graph in order to find all vertices that are reachable from the start vertex i by traversing paths from i to v for every $v \in i \rightarrow^*$.

To explore paths whose edge type sequence conforms to a regular path expression *rpe* in a search algorithm, the traversal of edges must be pre-checked according to the derived edge type sequence language defined by the expression *rpe*. Only paths whose derived edge type sequence conforms to *rpe* are to be allowed.

As cited in subsection 3.3 there is an accepting deterministic finite automaton $dfa(rpe)$ for every regular path expression *rpe* with $L(dfa) = L(rpe)$. Using Thompson's and Myhill's algorithms such an automaton can be computed easily.

rpe-reachability. Let *dfa* be a DFA for the edge type sequence of *rpe*, let s be some state in S_{dfa}, and let G be a graph with a given vertex $i \in V_G$. Then

$$reach_{G,dfa}(i, s) := \{v \in V \mid \exists\, C : Path \bullet \alpha(C) = i \wedge \omega(C) = v \wedge typeSeq(C) \in L(dfa, s)\}$$

is the set of all vertices reachable from i by a path whose derived edge type sequence is accepted by the state s in S_{dfa}.

Then, the problem of finding all vertices v which are reachable by a path conforming to some regular path expression *rpe* reduces to the derivation of all vertices in some set $reach_{G,dfa}(i, s_t)$, where $dfa = A(rpe)$ and $s_t \in F_{dfa}$ is a terminal state.

The simple algorithmic approach described above in Algorithm `SimpleSearch` for reachability problems can easily be generalized to solve *rpe*-reachability. To achieve this, the finite automaton $dfa(rpe)$ for $L(rpe)$ is used to guide the traversal procedure.

Instead of a boolean marking of the vertices, now the vertices are marked with the states of *dfa*, i.e. a vertex set marker

$$\texttt{marking} : Vertex \nrightarrow \mathbb{P}\, V_{dfa}$$

is assumed. Here, a vertex v gets marked by all states s such that $v \in reach_{G,dfa}(i, s)$.

```
1  interface VertexSetMarker<E> {
2    /** inserts value x into the set at vertex v */
3    void addValue (Vertex v, E x);
4
5    /** @return true iff the value x is in the set at vertex v */
6    boolean hasValue (Vertex v, E x);
7  }
```

Using this generalized marker, which assigns sets of automaton states to the vertices of the graph, a vertex v gets marked by a state s if and only if $v \in reach_{G,dfa}(i, s)$, i.e. if there is a path from i to v whose derived type sequence is accepted by s.

```
1  Algorithm: AutomatonDrivenSearch:
2  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3  // vertex−oriented search starting at vertex i guided by the automaton a
4
5  VertexSetMarker<State> m = new ...;
6  WorkList<Vertex x State> b = new ...;
7  VertexMarker<Edge x State> parentEdge = new ...;
8
9  void reach(Vertex i, Automaton dfa);
10   s := dfa.getStartState ();
11   m.addValue(i, s);
12   // process vertex i in state s;
13   b.insert(i,s);
14   while (! b.isEmpty()) {
15     // set(b, s) ⊆ {v ∈ V | v.isMarked(s)} ⊆ reach_{G,dfa}(i, s)
16     (v, s) := b.extractAny();
17     for (Edge e: v.getAllOutEdges()) {
18       // process edge e
19       w := e.getOmega();
20       for (Edge t: dfa.getAllEnabledTransitions(s, e.getType());
21         s1 := t.getOmega();
22         if (! m.hasValue(w,s1)) {
23           parentEdge.setValue((w,s1), (e,s));
24           m.addValue(w,s1);
25           b.insert(w,s1);
26           if (dfa.isTerminal (s1))
27             // process vertex w in state s1
28 } } } } }
```

The correctness arguments for Algorithm `AutomatonDrivenSearch` can be reduced to those of Algorithm `SimpleSearch`. Assume that the automaton *dfa* consists of the states $V_{dfa} = \{s_0, ..., s_k\}$. Then, Algorithm `AutomatonDrivenSearch` corresponds to a search on a *layered graph* (Figure 5) with the vertex set $V_G \times V_{dfa}$ where an edge from (v, s) to $(w, s1)$ exists if and only if

- there is an edge $e = v \rightarrow w$ in G and
- there is an edge $s \rightarrow_t s_1$ in A, where $t = type(e)$.

Algorithm `AutomatonDrivenSearch` terminates and visits exactly those vertices that are reachable from i via paths conforming to *rpe* in the layered graph.

If k is the number of states and l is the number of transitions of the automaton *dfa*, the inner loop (lines 20-28) is executed $l \times m$ times and its `if`-statement is executes $k \times n$ times at most, leading to an overall time complexity of $\mathcal{O}(max(k \times n, l \times m)$. Since in practice k and l are small multiples of the size of the regular expression *rpe* (Subsection 3.3) the algorithm is practically feasible.

Using a breadth-first approach, the algorithm delivers shortest paths also in this case. But it should be noted, that these are paths in the layered graph. In the orgininal graph, the paths themselves are not necessarily proper paths any more, i.e. vertices and edges may occur more than once on them.

Example. If the TGraph of Figure 3(a) is searched from A according to the GReQL path expression in `A (-->{a}-->{b})*-->{b}`, the automaton shown in Figure 4 can be used to drive the search. Here, s_0 is the start state, and s_2 is the (only) terminal state.

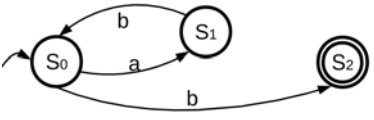
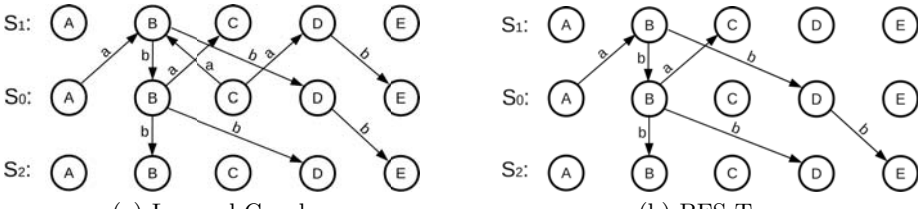


Fig. 4. DFA

Given this automaton, the resulting graph can be visualized using exactly one layer for each state (Figure 5(a)). Its breadth-first search tree is shown in Figure 5(b). Since s_2 is accepting, there are apparently three vertices in the result set $\{B, D, E\}$ and back tracing delivers three paths:

$$< A, 1, B, 2, B, 2, B >$$
$$< A, 1, B, 2, B, 5, D >$$
$$< A, 1, B, 5, D, 7, E >$$

All of these paths have minimum length, but only the third one is a proper path.



(a) Layered Graph (b) BFS Tree

Fig. 5. Search driven by automaton

Paths. The method to extract paths for each vertex found by Algorithm `Automaton-DrivenSearch` can also be generalized accordingly. Given a partial function

$$\texttt{parentEdge} : V_G \times V_A \nrightarrow E \times V_A$$

the paths can be enumerated, as well.

Assuming that s_t is a terminal state of A, and assuming that v is a vertex which is marked by s_t, i.e. $s_t \in m(v)$, a path can be computed by backward traversal in the layered graph:

```
process path i →* v:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  (z,s) := (v,s_t);
  // process vertex z in state s;
  while ((z,s) ≠ (i,s_0)) {
    (e,s) = z.parentEdge(z,s);
    // process edge e in state s;
    z = e.this();
    // process vertex z in state s
  }
```

Since *parentEdge* contains the complete information about all paths starting in i, this function is used as a representation of path systems in `JValue`.

6 Applications of Querying

The TGraph approach was developed in the course of tool development projects. Kogge [16] was a metaCASE tool, where TGraphs were used to realize an adaptable visual editor using a bootstrapping approach. Gupro [14] makes use of TGraphs as representations of software artefacts and supplies an integrated querying and browsing facility on code. Here, GReQL is the generic language used to extract information from source code. TGraphs together with GReQL are also used as the basic repository technology for artefact comparison and retrieval in ReDSeeDS [13].

Using a query language like GReQL, a lot of program analysis and program understanding tasks can be solved in a uniform manner. A query may deliver data for generic graph algorithms or can even replace a graph algorithm.

Examples for tasks that can be done using a query are

- complementing the graph by additional information that can be inferred, e.g., as a preprocessing step for other tasks,
- derivation of new views on the graph, e.g., derivation of the call graph, or
- computation of metrics.

In the following, we present some example GReQL queries to calculate some of this information on software represented as graphs according to the schema described in Figure 2 on page 338.

6.1 Edge Inference

The size and complexity of queries depends on the structure and completeness of the graphs and thus on the meta model and the parsers which are used for extracting the graphs from the source code.

Assume the parsers that extract the graph according to Figure 2 do not resolve all method invocations to their definitions but only the invocations of methods defined in the same class. Instead of computing the missing links in each query

where they are needed, the work on the graph can be simplified by querying this information once and *materializing* the result as edges in the graph.

GReQL itself is not meant to create edges, but it can be used to determine the respective set of vertex pairs which are to be connected. Then, the query is embedded into an algorithm that uses the query result and materializes the edges.

The query listed below finds the appropriate method declaration for every method invocation which is not yet properly linked to its declaration by the parser. Since GReQL is a declarative language, the query simply denotes which pairs are to be connected. (The GReQL optimizer assures an efficient evaluation of the query.)

```
1 from inv:V{MethodInvocation}, def:V{MethodDeclaration}
2 with isEmpty(inv <--{IsDeclarationOfInvokedMethod})
3  and theElement(inv <--{IsNameOfInvokedMethod}).name
4    = theElement(def <--{IsNameOfMethod}).name
5  and inv <--{IsMethodContainerOf}
6      ( <--{IsDeclarationOfInvokedMethod} <--{IsReturnTypeOf}
7      | <--{IsDeclarationOfAccessedVariable} <--{IsTypeOfVariable})
8      <--{IsTypeDefinitionOf}<--{IsClassBlockOf}<--{IsMemberOf} def
9 report inv, def end
```

The first and the last line state that pairs of `MethodInvocation` and `MethodDeclaration` vertices should be reported. The lines in between impose some constraints on the pairs to be reported by describing exactly the connection path between them. Line 2 excludes those invocations that are already linked to a declaration. Lines 3-4 force the names of the invoked method and the declaration to be equal. Method overloading is not considered here for reasons of brevity but can be included in the same way.

The most interesting part of the query is the path expression in lines 5-8. This path expression describes a path starting at the invocation and ending at the declaration. The first `IsMethodContainerOf` edge leads to the expression that returns the object on which the method is called. In the example `s.compareTo(data)` from page 337 the edge connects the call of `compareTo` to the variable access `s`. The following path alternative in lines 6-7 describes the path leading to the type of the object on which the method is called. The last part of the path in line 8 denotes the connection of this type with the method defined as a member of the main block of the class defining the type.

While the links are computed by the parser for invocations of methods defined in the same class, a proper linking of methods of other classes is possible only if the class whose method is called is known. For methods of objects which are returned by other method invocations or variable accesses, this class is obviously only known if the invocation or access is properly linked to the appropriate definition. This is a recursive problem which can either be solved in the query itself or by iterating the query in the embedding algorithm, which seems to be more elegant in this case.

6.2 Call Graph Computation

Similarly to the calculation of missing information that was not yet provided by the parser, it is also possible to calculate and materialize further information in the graph which enables a higher-level view. This is reverse engineering in the stronger sense, where more abstract information is derived from concrete data.

The query listed below computes the "call graph" of a given graph, i.e. it determines pairs of methods which call each other. This is done by looking at all method invocations contained in the body of a method. The query result can again be manifested as edges, in this case with edges of type `IsCalledByMethod`.

```
from caller, callee:V{MethodDeclaration}
with caller <--{IsBodyOfMethod} <--{IsStatementOf}*
            <--{IsDeclarationOfInvokedMethod} callee
report caller, callee end
```

6.3 Metrics Computation

Metrics constitute a quite natural application of querying. As an example, one of the metrics of the Chidamber&Kemerer metrics suite [10] is shown in the following: The CBO (coupling between object classes) metric assigns a natural number to each class, depicting the number of other classes it is coupled with. Here coupling means usage of variables or methods of the other class. Given a graph preprocessed as shown above, this metric can be calculated directly.

```
from c:V{ClassDefinition}
report
  c.name as "Class",
  count(
    c<--{IsClassBlockOf} <--{IsMemberOf}
    (<--{IsCalledByMethod}  |
     <--{IsBodyOfMethod,IsVariableCreationOf} <--{IsStatementOf}*
     <--{IsDeclarationOfAccessedVariable}-->{IsVariableCreationOf})
     -->{IsMemberOf} -->{IsClassBlockOf} &{thisVertex<>c}
  ) as "CBO"
end
```

For every class, the set of elements is computed that are reachable by a path that leads to an invocation of a method or an access of a variable of a different class. The size of this set is counted by the GReQL-function `count` and reported as the CBO for this class. The result of this query will be a table with two columns, named "Class" and "CBO" containing the class name and the number of classes this class is coupled with.

7 Related Work

Graphs as repository structures in software engineering tools and querying of these graphs is an enabling technology [24] in software reengineering. The definition of GXL [19] (which is inspired by TGraphs) as an interchange format for reengineering data gives further evidence that graphs are an appropriate abstraction used by many reengineering tools.

Graph Repositories. There are libraries for graphs which keep them in memory and provide a set of predefined algorithms on them, e.g., *LEDA* (The Library of Efficient Data Types and Algorithms) [26] and the *Boost Graph Library* (BGL) [33]. The latter provides different implementation variants for graph such as edge lists or adjacency lists and matrices.

GRAS [22] is a graph repository developed in the IPSEN project since 1985. A GRAS database is a graph pool that may contain several directed and typed graphs. (GRAS graphs have only vertex attributes and no edge ordering.) The structure of the graphs can be meta modeled by graph schemas and composition of graphs to hierarchical graphs is possible. GRAS supports direct main memory storage as well as storage in a relational database. *DRAGOS* (Database Repository for Applications using Graph Oriented Storage) [9], the successor of GRAS, is inspired by the graph exchange language GXL and overcomes the restrictions on edge attributes of GRAS. Additionally, DRAGOS allows the usage of directed hyper edges which may run between vertices as well as edges or endpoints of edges. Similarly to GRAS, DRAGOS also supports nesting of graphs.

JGraLab keeps the repository in memory and provides the modeling power of full TGraphs, which are kept in a data structure designed to support graph traversal efficiently. Edges are first class objects and may be traversed in any direction without additional costs.

Graph Query Languages. Apart from the XML query languages *XQuery* [8] and *XPath* [4] and the *SPARQL* RDF Query Language [31], there are also some languages that work on graph models directly. One such language is *Gram* [3] which includes walks and hyperwalks as a concept similar to paths and path systems. *GOQL* is an extension of the Object Query Language OQL used to query object-oriented databases enriched with constructs to query typed graphs. GOQL bears some syntactical analogy to GReQL as its queries are usually select-from-where statements and it allows to formulate simple path queries including constraints on the elements in a path. All these languages follow the idea of querying depicted in section 4, i.e. given a graph and a query, a value possibly containing graph elements is computed as the query result, which may also contain graph elements.

Languages like *GraphQL* [17] follow a different paradigm. Here, not only the data, but also the query and the query result are graphs. Then, query evaluation is a matching of graph patterns combined with graph rewrite rules, either transforming the given graph or creating a new one. Also *G+* [27] and its successor *GraphLog* [11] represent the query and the result as a graph and support regular path expressions and an evaluation by an automaton-driven search similar to

the one used in GReQL. These kinds of languages lead to graph transformation languages in general, like PROGRES [32], which may be interpreted as query languages in wider sense.

Compared to these languages, GReQL supports the most powerful form of regular path expressions, and only GReQL computes also paths and path systems as first class values.

Query languages for software re-engineering. Besides graph query languages, several other kinds of languages based on predicate logic or relational calculus are established in software re-engineering. [2] presents a detailed comparison of re-engineering languages based on their features. There are two main differences between all those languages and GReQL. The first refers to the representation of data, since edges as first class objects may directly represent occurrences of vertices in different places without artificial extra-nodes. The second is related to evaluation, which is search-based and does not compute and materialize relations, but explores only that part of the graph that is needed for query answering.

CrocoPat [5,6] is a relational programming language based on first-order predicate calculus. It uses Binary Decision Diagrams (BDDs) for internal storage of relations. Like many other tools in software reengineering, CrocoPat uses the Rigi Standard Format (RSF) to store relations in files. CrocoPat programs are written in the *RML* (Relation Manipulation Language) and consist of n-ary relational expressions. RML is an imperative language whose statements are executed sequentially, embedded in control structures such as `IF`, `FOR` and `WHILE`.

Supplying different ways to define n-ary relations and to combine existing relations to new ones, CrocoPat provides powerful means of data manipulation and retrieval. Regular path expressions can be simulated by logical operators to combine and by existential quantifiers to concatenate relations. E.g., the path expression

```
caller <--{IsBodyOfMethod} <--{IsStatementOf}*
       <--{IsDeclarationOfInvokedMethod} callee
```

used in Section 6.2 can be translated to the CrocoPat expression

```
calls(caller,callee) :=
       EX(body, IsBodyOfMethod(body,caller)
       & EX(statement, TC(IsStatementOf(statement,body))
       & IsDeclarationOfInvokedMethod(callee,statement))).
```

A language based on Tarski's relational calculus is *Grok* [18]. Grok is an untyped language, where all basic elements are represented as strings, but logical and mathematical operations can be applied to these elements. Sets and binary relations, which are just sets of tuples, are supported by Grok. Grok's operators generally apply across entire relations and not just to single entities.

The simulation of regular path expressions is possible by the concatenation of relations using the ∘ operator. As an example,

```
(inv IsBodyOfMethod) o (inv(IsStatementOf)*)
       o (inv IsDeclarationOfInvokedMethod)
```

represents the path expression used above. The result of applying this Grok operation is again a relation, which is stored to main memory to allow a fast access.

RScript [23] is statically typed in contrast to Grok. RScript reflects its primary application domain, the software analysis, in its features and data types. As an example, a data type `location` is provided, which represents a source locations as a combination of a filename and the position in the file. Besides the basic types `boolean`, `integer`, `string`, RScript provides the composite types tuples, sets and relations as well as user-defined types. Sets as well as relations can be nested and also a light version of n-ary relation is also supported in RScript. Similarily to Grok and CrocoPat, the combination of relations can be used to simulate regular path expressions.

All these languages look at graphs as sets of vertices and relations between vertices whereas GReQL has a focus on paths and traversals of graphs.

Comparison. While e.g. Alves et al. [2] have compared the main features of re-engineering query languages including the ones described above, there is no comparison of their performance up to now. Below, such a comparison is done exemplarily for CrocoPat and Grok, which seem to be the best established ones. and whose interpreters are publicly available. Two of the queries depicted in section 6, namely the calculation of the call-relation (Section 6.2) between methods and the CBO metrics (Section 6.3) are used for comparison. They were applied to the ASGs of four software systems, whose source code is freely available. Two small systems, the parser generator AntLR[2] and the test-environment JUnit[3] were used as well as two bigger systems, the Build-Tool Apache Ant[4] and the TGraph library JGraLab itself.

The abstract syntax graphs of the systems were extracted from the source code using a fact-extractor from Java to TGraphs, which makes use of edge attributes and ordering. The TGraphs were converted to the Rigi Standard Format (RSF), which can be imported by CrocoPat and Grok. Neither Grok nor CrocoPat allow for attributed relationships but require artificial relation-objects to represent such attributes. To keep the graphs and queries simple, we decided to use simple links instead of relation-objects and to accept, that the attributes of the edges were lost. While the attributes are not used in the queries below, they may be necessary for more complex analyses.

The evaluation times of both queries are shown in the table below for all three languages and all ASGs. Note, that CrocoPat and Grok are interpreters whose interpretation time is included in the overall result, whereas GReQL queries are parsed and optimized before evaluation. Since GReQL queries can be precompiled into a query library and reused for different graphs, the net evaluation time is given, as well.

The table shows that GReQL performs quite well in comparison to CrocoPat and Grok for the examples used.

[2] www.antlr.org
[3] www.junit.org
[4] ant.apache.org

	AntLR 90 classes 73k elements	**JUnit** 110 classes 42k elements	**Apache Ant** 1400 classes 1.1m elements	**JGraLab** 700 classes 1.7m elements
Calls-Query				
CrocoPat	1.9s	1.9s	62s	70s
Grok	0.3s	0.1s	7.0s	6.7s
GReQL	0.9s (0.13s eval)	0.8s (0.08s eval)	3.7s (2.7s eval)	3.5s (2.4s eval)
CBO-Query				
CrocoPat	2s	1s	2m 40s	1m 30s
Grok	1.0s	0.5s	9.6s	10.1s
GReQL	1.1s (0.16s eval)	1.1s (0.15s eval)	6.7s (5.8s eval)	3.8s (2.9s eval)

8 Conclusion

This paper showed how knowledge from graph algorithms can be used to construct efficient software engineering tools. It presented GReQL as an efficient and convenient graph query language. The efficient evaluation of regular path expressions in GReQL by search algorithms was explicated in detail.

GReQL can be used to query, enrich, abstract or analyze the graph representation of software engineering artifact. A few example applications in reverse engineering were shown. GReQL querying is also an enabling technology for software engineering tools in general since many kinds of information can be easily be extracted from graph-based models using queries [24], including information needed by the tool itself.

The usage of GReQL as a key technology in the reengineering tool GUPRO is presented in more detail in [14]. GUPRO is a generic tool that supports schema dependent browsing and querying of source code, using GReQL as its query language.

GReQL has been extended to include context-free path descriptions by Steffens [34]. Currently ongoing work aims at the extension of GReQL and the TGraph approach to more general distributed and hierarchical hyper-TGraphs (DHHTGraphs).

References

1. Aho, A., Sethi, R., Ullmann, J.: Compilers - Principles, Techniques and Tools. Addison-Wesley, Reading (1987)
2. Alves, T.L., Hage, J., Rademaker, P.: Comparative study of code query technologies (2009), Online PDF (04.06.2010), `wiki.di.uminho.pt/twiki/pub/Personal/Tiago/Publications/Alves09b-draft.pdf`

3. Amann, B., Scholl, M.: Gram: A graph data model and query language. In: European Conference on Hypertext (1992)
4. Berglund, A., et al.(eds.): XML Path Language (XPath) 2.0, W3C Recommendation (January 2007)
5. Beyer, D.: Relational programming with crocopat. In: ICSE 2006: Proceedings of the 28th International Conference on Software Engineering, pp. 807–810. ACM, New York (2006)
6. Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. IEEE Trans. Softw. Eng. 31(2), 137–149 (2005)
7. Bildhauer, D.: Entwurf und Implementation eines Auswerters für die TGraphanfragesprache GReQL 2. VDM Verlag (2008)
8. Boag, S., et al. (eds.): XQuery 1.0: An XML Query Language, W3C Recommendation (January 2007)
9. Böhlen, B.: Ein parametrisierbares Graph-Datenbanksystem für Entwicklungswerkzeuge. Shaker Verlag, Aachen (December 2006)
10. Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object oriented design. In: OOPSLA 1991: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications, pp. 197–211. ACM, New York (1991)
11. Consens, M.P., Mendelzon, A.O.: Graphlog: a visual formalism for real life recursion. In: PODS 1990: Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 404–416. ACM, New York (1990)
12. Ebert, J.: A Versatile Data Structure For Edge-Oriented Graph Algorithms. Communications ACM 30(6), 513–519 (1987)
13. Ebert, J., Bildhauer, D., Riediger, V., Schwarz, H.: Using the TGraph Approach for Model Fact Repositories. In: Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008) (May 2008)
14. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO. Generic Understanding of Programs - An Overview. Electronic Notes in Theoretical Computer Science 72(2) (2002), `http://www.elsevier.nl/locate/entcs/volume72.html`
15. Ebert, J., Riediger, V., Winter, A.: Graph Technology in Reverse Engineering, the TGraph Approach. In: Gimnich, R., et al. (eds.) 10th Workshop Software Reengineering (WSR 2008), Bonn. GI Lecture Notes in Informatics, vol. 126, pp. 67–81. GI (2008)
16. Ebert, J., Süttenbach, R., Uhe, I.: JKogge: a Component-Based Approach for Tools in the Internet. In: Proceedings STJA 1999, Erfurt (1999)
17. He, H., Singh, A.K.: Graphs-at-a-time: query language and access methods for graph databases. In: SIGMOD 2008: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 405–418. ACM, New York (2008)
18. Holt, R.C.: Wcre 1998 most influential paper: Grokking software architecture. In: WCRE, pp. 5–14 (2008)
19. Holt, R.C., Schürr, A., Sim, S.E., Winter, A.: GXL: a graph-based standard exchange format for reengineering. Science of Computer Programming 60(2), 149–170 (2006)
20. Hopcroft, J.E., Ullmann, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
21. Horn, T.: Ein Optimierer für GReQL2. GRIN Verlag GmbH (2009)
22. Kiesel, N., Schürr, A., Westfechtel, B.: GRAS, a graph oriented (software) engineering database system. Information Systems 20(1), 21–51 (1995)

23. Klint, P.: Using Rscript for software analysis. In: Working Session on Query Technologies and Applications for Program Comprehension (QTAPC) (2008)
24. Kullbach, B., Winter, A.: Querying as an Enabling Technology in Software Reengineering. In: Verhoef, C., Nesi, P. (eds.) Proceedings of the 3rd Euromicro Conference on Software Maintenance & Reengineering, pp. 42–50. IEEE Computer Society, Los Alamitos (1999)
25. Marchewka, K.: GReQL 2. Master's thesis, Universität Koblenz-Landau, Institut für Softwaretechnik (2006)
26. Mehlhorn, K., Näher, S., Uhrig, C.: The leda platform of combinatorial and geometric computing. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 7–16. Springer, Heidelberg (1997)
27. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM Journal on Computing 24(6), 1235–1258 (1989)
28. Myhill, J.R.: Finite automata and the representation of events. Technical Report 57-624, Wright Patterson AFB, Ohio (1957)
29. Nagl, M.: An incremental compiler as component of a system for software generation. In: Programmiersprachen und Programmentwicklung. Fachtagung des Fachausschusses Programmiersprachen der GI, vol. 6, pp. 29–44. Springer, London (1980)
30. Object Management Group: Meta Object Facility (MOF) Core Specification, OMG Availiable Specification, Version 2.0 (2006)
31. Prud'hommeaux, E., Seaborne, A. (eds.): SPARQL Query Language for RDF, W3C Recommendation (January 2008)
32. Ranger, U., Weinell, E.: The graph rewriting language and environment PROGRES. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 575–576. Springer, Heidelberg (2008)
33. Siek, J.G., Lee, L., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, Reading (2001)
34. Steffens, T.: Kontextfreie Suche auf Graphen. VDM Verlag (2008)
35. Thompson, K.: Regular expression search algorithms. Communications of the ACM 11(6), 419–422 (1968)

Graph-Based Structural Analysis for Telecommunication Systems

André Marburger[1] and Bernhard Westfechtel[2]

[1] DSA Daten- und Systemtechnik GmbH, Pascalstr. 28, D-52076 Aachen
andre.marburger@gmx.de
[2] Applied Computer Science I, University of Bayreuth, D-95440 Bayreuth
bernhard.westfechtel@uni-bayreuth.de

Abstract. Many methods and tools for the reengineering of software systems have been developed so far. However, the domain-specific requirements of telecommunication systems have not been addressed sufficiently. The E-CARES project is dedicated to reverse engineering of complex legacy telecommunication systems by providing graph-based tools. With E-CARES, the software architecture of a telecommunication system is recovered in two steps. In the first step (program analysis), the source code is parsed to build a structure graph which uses the abstractions of the underlying programming language and describes the internals of program units (blocks) as well as their communication via exchange of signals. In the second step, a software architecture description is abstracted from the structure graph. The software architecture is described in ROOM, a real-time object-oriented modeling language for embedded systems design. Both program analysis and architecture recovery are based on graphs and graph transformations. In both steps, domain-specific knowledge — referred to as methods of use — is exploited which refers to the ways how language constructs are used to realize processing concepts of telecommunication systems.

Keywords: Reverse Engineering, Structural Analysis, Telecommunication System, Graph Transformation.

1 Introduction

Reengineering of large and complex software systems has proved a difficult task. According to the "horseshoe model of reengineering" [1,2], reengineering is divided into three phases. *Reverse engineering* is concerned with step-wise abstraction from the source code and system comprehension. In the *restructuring* phase, changes are performed on different levels of abstraction. Finally, *forward engineering* introduces new parts of the system (from the requirements down to the source code level).

For reengineering, many methods and tools have been developed. To a large extent, however, previous work has been *data-centered* since it focuses on structuring the data maintained by an application. In particular, numerous approaches have addressed the migration of legacy business applications — written, e.g., in

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 363–392, 2010.

COBOL — to an object-based or object-oriented architecture [3,4]. This task requires the grouping of data and functions into classes with corresponding attributes and methods. Another stream of research has dealt with programming languages such as C++ and Java which already provide language support for object-oriented programming [5].

Reengineering of *process-centered* applications has been addressed less extensively so far [6]. For example, a telecommunication system is composed of a set of distributed communicating processes which are instantiated dynamically for handling calls requested by its users. Such a system is designed in terms of services provided by entities which communicate according to protocols. Understanding a telecommunication system requires the recovery of this conceptual world from the actual source code and other sources of information.

The *E-CARES*[1] research cooperation between Ericsson Eurolab Deutschland GmbH (EED) and Department of Computer Science III, RWTH Aachen University, has been established to develop methods, concepts, and tools for the reengineering of complex legacy telecommunication systems. E-CARES has been driven strongly by the requirements of software engineers who are involved in the design and implementation of GSM networks for mobile communication. The object of study is Ericsson's Mobile-service Switching Center (MSC) for GSM networks called AXE10. The AXE10 software system comprises approximately 10 million lines of code spread over about 1,000 executable units, and has an estimated lifetime of about 40 years. Thus, there is an urgent need for tool support to improve program evolution and maintenance.

This paper deals with the *reverse engineering environment* developed within the E-CARES project, as presented comprehensively in [8]; see [9] for restructuring. Moreover, we confine the presentation to *structural analysis*, i.e., recovery of the structure of the system under study. For the sake of modularity and reuse, structural analysis is decomposed into two major steps. The first step, which builds a suitable abstraction of the source code, is called *program analysis*. The resulting representation still depends on the underlying programming language (PLEX, a proprietary language which was developed at Ericsson in the 1970s). In contrast, the second step — *architecture recovery* — delivers a representation in a modeling language which is not specific to the programming language any more. For architecture recovery, we selected *ROOM* [10] as the target language, since it was applied extensively to the development of telecommunication systems and — by and large — provides appropriate concepts and abstractions for this domain (see [11] for further extensions to ROOM to improve the modeling of telecommunication systems). However, we could have alternatively used another language, e.g., SDL [12] or the UML component model [13].

To recover meaningful abstractions of the program source code, *domain-specific knowledge* is exploited heavily in structural analysis. In the context of the E-CARES project, this knowledge was summarized under the term *methods of use*: Telecommunication experts at Ericsson use the programming language and the runtime system in systematic ways in order to realize processing concepts

[1] *E*ricsson *C*ommunication *AR*chitecture for *E*mbedded *S*ystems [7].

and paradigms. For example, the AXE-10 system follows a signaling paradigm of processing, where instances of blocks (program units in PLEX) cooperate to operate calls by passing signals along a link chain between the originator of a call and its receiver. The way how the signaling paradigm is mapped onto PLEX results in domain-specific code patterns which are subsequently transformed into design patterns at the architectural level.

The E-CARES reverse engineering environment is *graph-based*, since recovered structures are represented as graphs. Graphs serve as a general, natural, and expressive data model which is frequently used in many other reengineering environments, as well. However, E-CARES differs from other reengineering environments because it relies on *graph transformations*: Large parts of the E-CARES environment are generated from a formal specification based on a graph transformation system. In this way, the effort for developing the reverse engineering environment has been reduced significantly.

The remainder of this paper is structured as follows: Section 2 provides some background knowledge from the telecommunication domain. Section 3 gives a brief overview of the E-CARES environment. Sections 4 and 5 are devoted to program analysis and architecture recovery, respectively. Section 6 is concerned with the realization of the E-CARES environment. Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 Background

2.1 GSM Basics

The *mobile-service switching centers* are the heart of a GSM network (Figure 1). An MSC provides the services a person can request by using a mobile phone, e.g., a simple phone call, a phone conference, or a data call, as well as additional infrastructure like authentication. Each MSC is supported by several Base Station Controllers (BSC), each of which controls a set of Base Station Transceivers (BTS). The interconnection of MSCs and the connection to other networks (e.g., public switched telecommunication networks) is provided by gateway MSCs (GMSC). In fact, the MSC is the most complex part of a GSM network. An MSC consists of a mixture of hardware (e.g., switching boards) and software units. In our research we focus on the software part of this embedded system.

Figure 2 illustrates how a *mobile originating call* is handled in the MSC. The figure displays logical rather than physical components according to the GSM standard; different logical components may be mapped onto the same physical component. The mobile originating MSC (MSC-MO) for the A side (1) passes an initial address message (IAM) to a GMSC which (2) sends a request for routing information to the home location register (HLR). The HLR looks up the mobile terminating MSC (MSC-MTE) and (3) sends a request for the roaming number. The MSC-MTE assigns a roaming number to be used for addressing during the call and stores it in its visitor location register (VLR, not shown). Then, it (4) passes the roaming number back to the HLR which (5) sends the

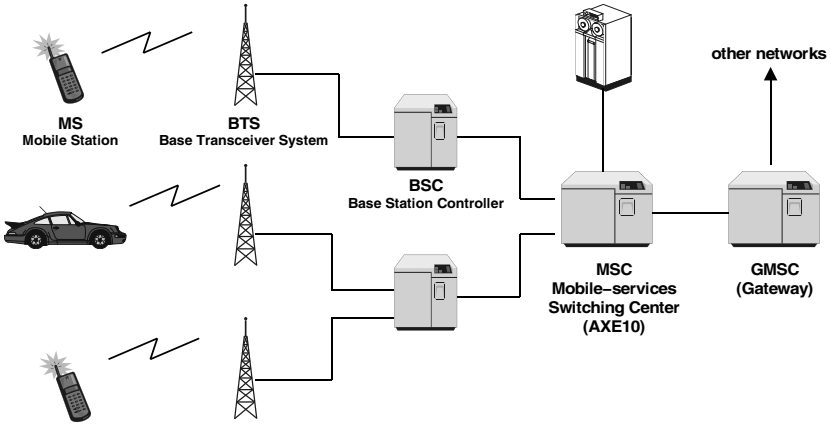


Fig. 1. Simplified sketch of a GSM network

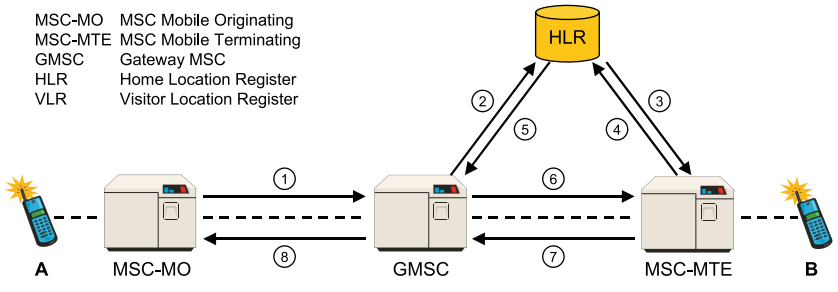


Fig. 2. Mobile originating call

requested routing information to the GMSC. After that, the GMSC (6) sends a call request to the MSC-MTE. The MSC-MTE (7) returns an address complete message (ACM) which (8) is forwarded to the MSC-MO. Now, user data — e. g., the two participants' voices — may be transferred between A and B as indicated by the dashed line in the figure.

2.2 The Programming Language PLEX

The application part of the AXE10 software is implemented in a non-standard programming language called PLEX (**P**rogramming **L**anguage for **EX**changes), which was developed at Ericsson. PLEX is an asynchronous concurrent real-time language designed for programming of telecommunication systems. This programming language has a "signaling paradigm" as the top execution level. That is, only events can trigger code execution. Events are programmed as signals.

A PLEX program is composed of a set of *blocks* (designated by the keyword DOCUMENT) which serve as units of compilation. Blocks have data encapsulation, that is, a block's data cannot be accessed by other blocks. PLEX has a COBOL-like syntax with blocks being divided into multiple sectors. Only two of these are of interest within the scope of this paper, namely the *declare sector* for the declaration of local variables, and the *program sector* for the control logic. As an example, Figure 3 shows cutouts of several blocks being involved in the processing of a mobile originating call as illustrated in Figure 2.

The *declare sector* defines variables for elementary data types such as e.g. integers, strings, and symbols, as well as structured data such as arrays and records. Data may be marked as persistent by a corresponding keyword in the declaration. A *record* which is marked as persistent stands for a file storing instances of records of this type. Typically, a record instance holds data for a specific call. For example, the record MSCMO in block MSCMO holds — among other data — the telephone number of the telephone of the B side, the so-called BNUMBER.

Blocks communicate via *signals* which are declared in separate signal description tables (not shown). The SEND statement serves to send a signal. In the case of a *unique signal*, the recipient is not specified as it may be identified by a simple name match of the signal name in the reception statements of other blocks. In the case of a *multiple signal*, the SEND statement contains a reference to a variable which stores the recipient at runtime. Here, the name match alone is ambiguous as it only provides possible pairs of senders and receivers. Data may be transferred along with a signal via the WITH clause. By default, signals realize asynchronous communication (*single signals*). In the case of synchronous communication (*combined signals*), the SEND statement specifies the set of alternative signals which are expected as backward signals (WAIT FOR clause). Single and combined (forward) signals are received by ENTER and RECEIVE statements, respectively. In the latter case, backward signals are sent via RETURN statements and received by RETRIEVE statements.

The control flow of a block is described in its program sector. Execution starts with the reception of a signal and continues until an EXIT statement is reached. Typically, a GOTO statement is executed in response to a signal which transfers the control flow to a *labeled statement sequence*. A DO statement calls a parameter-less *subroutine* which is embraced by BEGIN and END statements. Parameters can be passed only with *local signals* which are processed by the sending block itself. This is achieved via a TRANSFER statement which performs an unconditional jump and optionally passes parameters to the target code fragment (starting with an ENTRANCE statement).

As presented so far, a block is a fairly unstructured program unit. Unfortunately, PLEX does not offer language constructs to decompose blocks into more fine-grained logical units. At a conceptual level, however, such units do exist (and need to be recovered by structural analysis). First, since execution is driven by signals, the code of the program sector may be decomposed into *functions*, i.e., pieces of code being executed in response to some signal. Second,

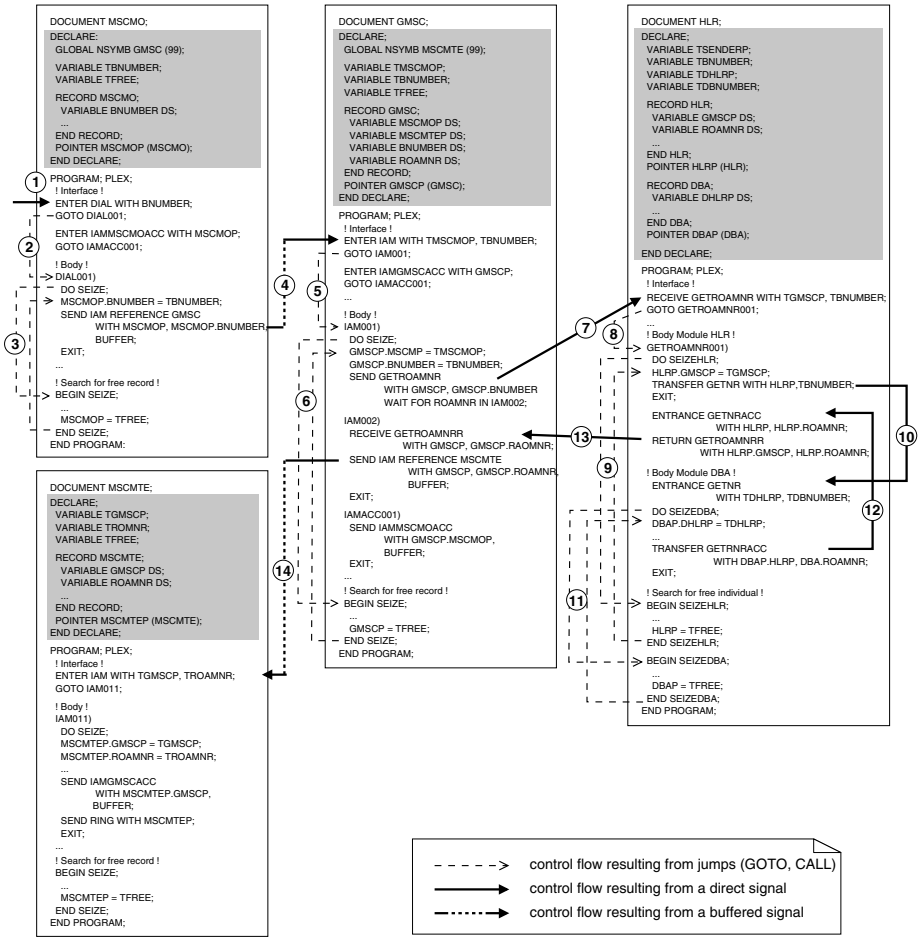


Fig. 3. PLEX blocks and their communication

multiple functions may be aggregated into *modules*, which provide logically coherent "mini-services". For example, the block HLR consists of two modules — the main module HLR and the subordinate module DBA (data base access). The latter serves as a wrapper that provides a standardized interface to a specific physical database to decouple the application from a specific database implementation.

2.3 Execution Model

Program systems written in PLEX must be run on top of an operating system whose main task is signal handling. Single signals are managed in priority queues called *job buffers*. The runtime system selects a signal from a queue and delivers it

to the receiving block. *Direct signals* (for local and synchronous communication) are executed immediately rather than stored in job buffers. Within a block, memory is usually managed statically. If resources are exhausted, further space may be allocated via size alteration events.

At runtime, telecommunication systems handle thousands of different telephone calls at the same time. Each call — as e.g. illustrated in Figure 2 — represents a service request to the telecommunication system. The numerous services are realized by re-combining and re-connecting sets of small services, provided by respective blocks. Thus, a telephone call is realized by numerous mini-services spontaneously linked together. Telecommunication experts call the interconnected services for a single telephone call a *link chain*.

Conceptually, a link chain is composed of a set of interconnected *block instances*, each of which holds data for a specific call. Since PLEX does not provide language support for dynamic instantiation of blocks, block instances have to be managed "manually" by the respective block, i.e., the PLEX programmer has to simulate dynamic instantiation by writing code for allocating and deallocating memory units, etc. If a block consists of multiple modules, instantiation has to be performed at the module level, resulting in module instances. For the sake of simplicity, however, we will use the term "block instance" in the description below.

Figure 3 shows a cutout of the processing of a mobile originating call as an example for the dynamic creation of a link chain. The figure covers part of the mobile originating call scenario of Figure 2, namely those steps in which the blocks MSC-MO, GMSC, and HLR are involved. Processing starts with the reception of a DIAL event (1), which causes a jump to a labeled statement sequence for processing this event (2). An idle MSC-MO record is occupied through the call of the subroutine SEIZE (3). After that, the signal IAM is sent to block GMSC (step (4), which corresponds to step 1 in Figure 2). The recipient in turn jumps to a labeled statement sequence (5) and allocates a record for the link chain (6). Subsequently, it sends a request for a roaming number to block HLR (step 7, corresponding to (2) in Figure 2). After a jump to the processing labeled statement sequence (8) and seizing of a record for the new call (9), a local signal is sent (10) which triggers the allocation of another record in the subordinate module DBA (11). Then, another local signal is sent (12) whose associated data contains the roaming number for the call. The roaming number is passed back to GMSC (step 13, corresponding to (5) in Figure 2), from which it is finally forwarded to MSC-MO (step 14, corresponding to (8) in Figure 2).

3 The E-CARES Environment

Within the E-CARES project, a *reengineering environment* for telecommunication systems has been developed. The system architecture of the E-CARES environment is illustrated in Figure 4. In this paper, we are concerned only with reverse engineering (solid lines). The components displayed with dashed lines deal with restructuring and forward engineering; see [9].

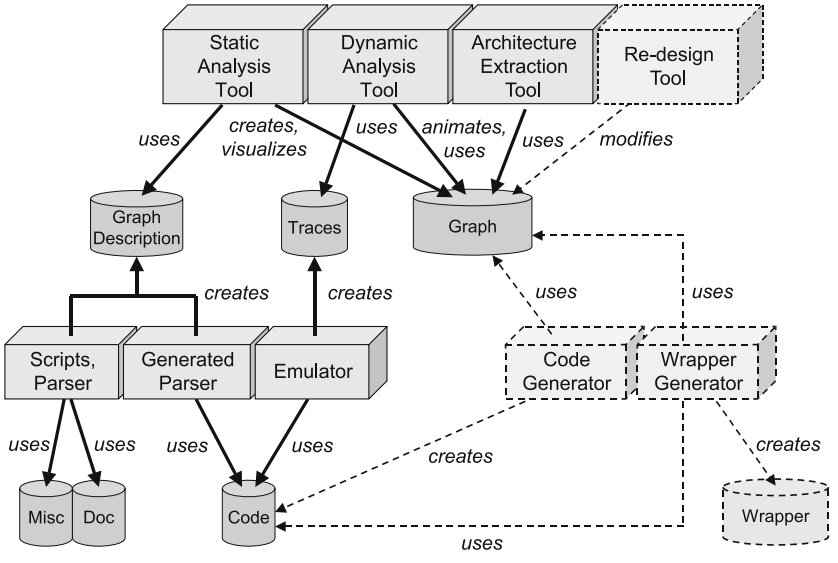


Fig. 4. Architecture of the E-CARES environment

Reverse engineering involves different kinds of analysis: *Structural analysis* — the focus of this paper — refers to the static system structure, while *behavioral analysis* is concerned with the system's dynamic behavior. Thus, the attributes "structural" and "behavioral" denote the outputs of analysis. In contrast, *static analysis* denotes any analysis which can be performed on the source code (or structure document, respectively), while *dynamic analysis* requires information from program execution. Thus, "static" and "dynamic" refer to the inputs of analysis.

For the static analysis of the structure of a PLEX system, we base our system on three sources of information. The first one is the *source code* of the system. It is considered to be the core information as well as the most reliable one. Via code analysis (parsing) a number of structure documents is generated from the source code, one for each block. These structure documents form a kind of textual graph description. The second and the third source of information are *miscellaneous documents* (e.g., product hierarchy description, signal database) and the system *documentation*. As far as the information from these sources is computer processable, we use parsers and scripts to extract additional information. This additional information is stored in structure documents, as well.

The *static analysis tool* processes the graph descriptions of individual blocks and extends the structure graph which represents the overall application by creating corresponding subgraphs. The subgraphs are then connected by performing global analyses in order to bind signal send statements to signal entry points. Moreover, the subgraphs for each block are reduced by performing simplifying graph transformations. The static analysis tool also creates views of the system at different levels of abstraction. In addition to structure, static analysis is

concerned with behavior (e.g., extraction of state machines or of potential link chains from the source code).

The graph produced by the static analysis tool depends on the programming language (PLEX) in which the source code is written. Therefore, we will use the term *program analysis* (Section 4) to denote this step of processing. The *architecture extraction tool* transforms the structure graph into elements of a modeling language which does no longer use abstractions specific to the programming language. This results in an architecture graph that can be used to review and (re-)document the currently implemented architecture or to perform architectural changes. As modeling language, we selected ROOM; other languages — e.g., SDL or the UML component model — could be supported in an analogous way (see also Section 7 on related work).

Dynamic information originates from using an emulator or querying a running AXE10, resulting in a list of events plus additional information in a temporal order in both cases. Such a list constitutes a *trace* which is fed into the *dynamic analysis tool*. Interleaved with trace simulation, dynamic analysis creates a graph of interconnected block instances that is connected to the static structure graph. This helps software architects to identify components of a system that take part in a certain traffic case. At the user interface, traces are visualized by collaboration and sequence diagrams. In E-CARES, dynamic analysis is performed only for recovering the behavior; therefore, it goes beyond the scope of this paper. For further information on dynamic analysis, the reader is referred to [8,14].

4 Program Analysis

Program analysis builds a structure graph which still depends on the underlying programming language PLEX. However, program analysis does not exploit only the syntactic structure of PLEX programs. In addition, it takes advantage of *methods of use*, i.e., coding conventions and programming patterns which are applied by software developers at Ericsson. These methods of use constitute indispensable expert knowledge which assists in building program representations with appropriate domain-specific abstractions.

4.1 Building the Structure Graph

The *structure graph* is created by parsing diverse sources. The most important information is the source code of PLEX blocks. In addition, global signals are declared in signal description tables. Furthermore, information about the coarse-grained system structure above the level of blocks is given in product configuration files. In the following, we will primarily focus on parsing of PLEX blocks.

The design of the structure graph was driven heavily by the requirements of telecommunication experts at Ericsson. From the very beginning, the experts were more interested in the coarse-grained structure of the system under study rather than in detailed code analysis. Therefore, the structure graph contains

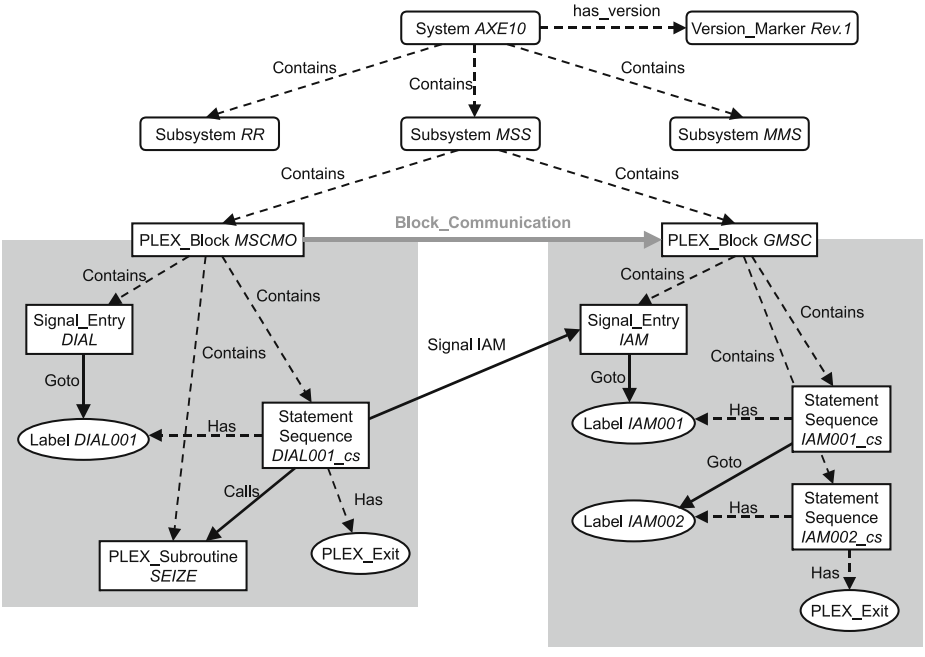


Fig. 5. Example of a structure graph

only the information which is required for the purposes of reverse engineering. In particular, the structure graph has to reveal the interfaces of blocks (processed signals), the actions performed in response to received signals, and the inter-block communication.

For this reason, the structure graph is more compact than an *abstract syntax graph*, which would provide a detailed syntactic representation of the source code. From the structure graph alone, it would not be possible to reconstruct the source code. However, the structure graph contains references to text lines in the source code. Thus, for each element of the structure graph the source code fragment may be retrieved from which this element was derived.

An example of a (cut-out of a) structure graph is presented in Figure 5. Partially, this structure graph is created from the code fragments shown in Figure 3. The nodes represented by rounded rectangles on the top do not correspond to language constructs of PLEX. Rather, they are created by parsing a configuration file which decomposes the AXE10 system into subsystems and blocks (as leaves of the composition hierarchy).

The PLEX parser processes each block independently, resulting in a block graph (shaded regions) which is embedded into its context in a subsequent phase. The figure shows only parts created from the program sector. In addition, a block graph contains further elements created from other sectors, e.g., nodes for variable declarations.

The program sector is represented in such a way that the computational behavior of blocks is reflected. Thus, it contains nodes for *signal entries*, which constitute the export interface of a block. The coding conventions at Ericsson state that a GOTO statement — represented by an edge — is executed in response to the signal reception. Its target is a *label* (nodes displayed as ellipses) which marks the start of some *statement sequence*. Within this sequence, subroutines may be called. Furthermore, a statement sequence may contain an EXIT statement which returns control to the dispatcher of the operating system.

The notion of *labeled statement sequence* constitutes an example of the methods of use mentioned above. A labeled statement sequence does not correspond to a syntactic unit of the PLEX programming language. Rather, a labeled statement sequence is defined as a code fragment starting with some label and ending before the next label. Labeled statement sequences constitute a useful abstraction which reflects the way in which PLEX programs operate: They execute statement sequences in response to signals. The internals of these sequences are not important; therefore, they are not represented in the structure graph.

After building initial subgraphs for each block, several steps of *postprocessing* have to be performed. For example, the *control flow* has to be completed by postprocessing. Like in COBOL, execution of a labeled statement sequence may *fall through* to a consecutive statement sequence if the preceding sequence is not terminated by an EXIT statement. In this case, the labeled statement sequences are connected by a control flow edge. This kind of processing is a prerequisite for the recovery of functions to be discussed in the next subsection.

Furthermore, after initial local analysis blocks are isolated and need to be connected by *binding* global signal sending statements to signal entry points. Local analysis prepares the binding by creating virtual signal entry points acting as temporary target nodes (not shown in Figure 5). Global analysis searches for matching entry points in other blocks, creates inter-block edges for representing the sending of signals, and removes the temporary virtual signal entry points. For example, this binding step inserts an edge from the labeled statement sequence DIAL001 in block MSCMO to the signal entry point IAM in block GMSC. In the case of single signals, the binding is unique. In the case of multiple signals, the actual receiver is determined only at run time; static analysis creates an edge to the entry point of each potential receiver.

In the binding step, it is crucial to take the following method of use into account: In AXE10, each block is responsible for returning control to the operating system within a maximal time slice. For efficiency reasons, there is no interrupt handling built into the operating system. But the processing of an incoming signal may consume more than one time slice. Thus, the processing is divided into multiple phases where each but the last phase sends a CONTINUE signal to the block itself and terminates. This mode of processing is known as *phase division*.

CONTINUE signals have to be excluded from the binding step; otherwise, the structure graph would be cluttered with many erroneous edges (recall that in the case of multiple signals edges are created to all potential receivers). Rather, these signals are handled specially by a phase division analysis. All phases which

are executed in response to a certain signal are connected by control flows. In this way, artificially separated computation fragments are joined together.

After having created connections between signal sending statements and single entry points, the communications are *lifted* to the block level. For example, lifting results in the block communication edge from MSCMO to GMSC. Furthermore, lifting is continued to the subsystem level by connecting subsystems with mutually communicating blocks.

4.2 Recovering Functions and Modules

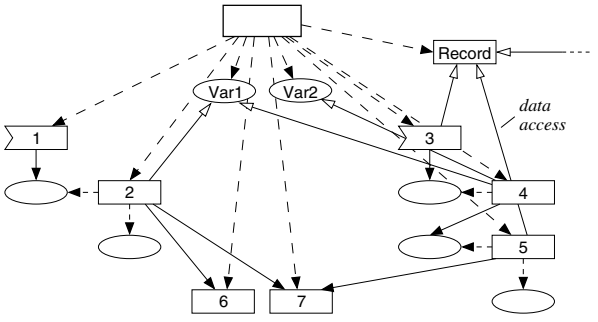
Functions and modules are conceptual abstractions rather than syntactic units of the PLEX programming language. Several methods of use are exploited to recover these abstractions from the structure graph built up so far. Since the recovery of modules is concerned with the grouping of functions, the recovery of functions is described first.

Recovery of Functions. A *function* represents the set of code fragments which are uniquely executed as a reaction to the reception of a certain signal. Consequently, each function contains exactly one signal entry. The remaining nodes of the structure graph that belong to a function are determined through control flow analysis.

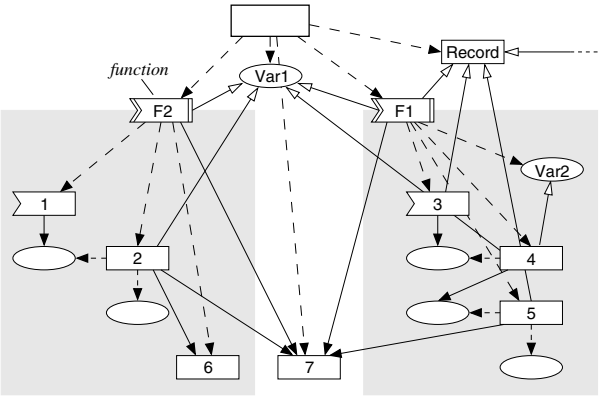
Figure 6 illustrates the recovery of functions (and modules, see below) by an example. In this example, we start with a block structure graph comprising two signal entries (nodes 1 and 3), three labeled statement sequences (nodes 2, 4, and 5), and two subroutines (nodes 6 and 7). Furthermore, some data elements, one record and two atomic data elements (Var1 and Var2), are depicted. The further structuring of the record has been omitted for the sake of brevity. Therefore, the data access edges, represented by arrows with open heads, normally accessing some data element in the record have been redirected to the record node.

For function recovery, all graph elements are determined that can be reached from the given signal entry via a transitive closure of control flow edges. The preceding postprocessing steps ensure that the control flow is represented completely in the structure graph; consider e.g. processing of fall-through execution and phase divisions as explained in the previous subsection. With respect to the example in Figure 6, the transitive closure returns a set consisting of nodes 2, 6, and 7 when starting at signal entry 1. Furthermore, all data elements accessed from these nodes are determined. For example, in the case of signal entry 1 the variable Var1 is accessed. Starting from signal entry 3, we obtain the variables Var1 and Var2 as well as the record.

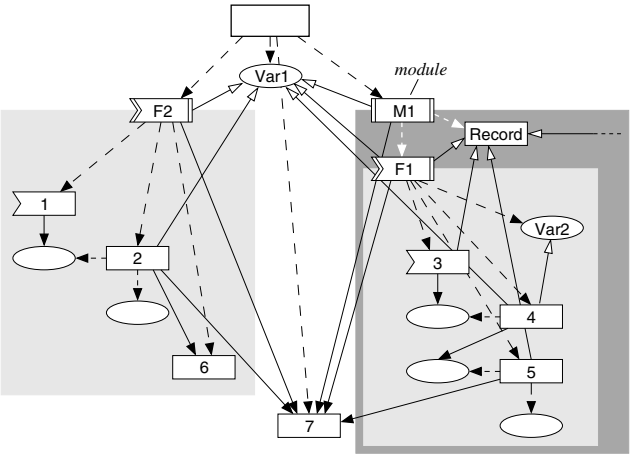
Next, graph elements shared by at least two transitive closures are removed from the respective sets in order to obtain only uniquely assignable graph elements. In Step 2 of Figure 6, all uniquely assignable graph elements are shown in regions with a light gray background. The subroutine 7, the variable Var1, and the record — which is also accessed from another function not displayed in the figure — are located outside these regions because they are shared by multiple



Step 1: Structure graph after embedding, correction, and completion



Step 2: Functions detected and created



Step 3: Module detected and created

Fig. 6. Effects of detection of functions and modules on a structure graph

functions. On the other hand, subroutine 6 and variable Var2 have been identified as local elements.

The graph shown in Step 2 also includes the function nodes (F1 and F2) which have been created to represent the results of the recovery in the structure graph. These nodes are inserted in between the block node and the nodes representing local graph elements. Furthermore, outgoing control flow and data flow edges have been lifted to the function node. In this way, the "imports" of each function are represented.

Recovery of Modules. Recovery of functions serves to identify the *interface* of a block. As already mentioned at the end of Subsection 2.2, blocks may be decomposed into more fine-grained units called *modules*. A module groups data and functions. For example, the block HLR consists of two modules — the main module HLR and the subordinate module DBA (data base access).

As we have explained in Subsection 2.3, telephone calls are handled by building up link chains. A link chain is composed from instances of participating blocks. If a block is decomposed into multiple modules, instantiation has to be performed on the level of modules. For each instance, its data are typically held in a single record.

This domain knowledge is exploited in *module recovery*. In contrast to function recovery, which is based on control flow analysis, module recovery analyzes accesses to data elements. However, we do not consider all data elements declared in a block. Rather, we confine the scope of analysis to records holding instance data. These records are discovered by applying some heuristic rules described below.

A record indicating that there is a module contained in a PLEX block satisfies certain requirements. First of all, instance records are stored records. They are stored in a file and are referenced by a similarly named pointer variable. In addition, instance records have to contain an enumeration type variable defining a list of symbolic state values. According to a coding rule, the name of this variable contains the substring STATE. Therefore, a string match is sufficient for its identification. Another good hint that a record is an instance record is its size. Instance records tend to be much larger than other records in a block.

Since modules are grouped around an instance record, we use a data centered algorithm for the calculation of the module candidate. A well-known algorithm of this class has first been proposed by Liu and Wilde in [15]. An adaption of this algorithm to graphs can be found in [16]. Originally, the idea of the algorithm is to identify *object candidates* in procedural code. The idea bases on the observation that software has already been developed in an object-based manner long before object-oriented design has become popular. Thus, it is assumed that object candidates in a conventional programming language can be identified as a collection of routines, (global) data types and/or (global) data elements. That is, an object candidate is a triple (F, T, D), where F is a set of functions, T a set of data types, and D a set of data elements.

Adapted to the facts of E-CARES, object candidates conform to module candidates. Furthermore, the set of data types will always be empty as there are

no user defined data types in PLEX. Also, the data elements used as an input to the algorithm can be limited to records only. Therefore, the procedure of the module candidature for PLEX code according to Liu and Wilde reads as follows:

1. *For each instance record x, let $P(x)$ be the set of routines which directly use x. Moreover, x should be shared by at least two different routines. Routines can correspond to functions (if present[2]) but also to any other kind of multi-statement structure object inside a block (signal entry, statement sequence, subroutine).*
2. *Considering each $P(x)$ as a node, construct a graph $G = (V, E)$ such that*

$$V = \{\ P(x) \mid x \text{ is shared by at least two routines}\} \quad \text{is a set of nodes,}$$

$$E = \{\ \overline{P(x_1)P(x_2)} \mid P(x_1) \cap P(x_2) \neq \emptyset\} \quad \text{is a set of undirected edges.}$$

3. *Construct a module candidate (F, T, D) from each strongly connected component $G_{cand} = (V_{cand}, E_{cand})$ in G such that*

$$F = \bigcup_{P(x) \in V_{cand}} P(x)$$

$$T = \emptyset$$

$$D = \bigcup_{P(x) \in V_{cand}} \{x\}$$

4. *Complete each module candidate by adding all data elements which are accessed only locally.*

Applied to the graph shown in Step 2 of Figure 6, the algorithm selects the only record as its starting point (assuming that the heuristic rules described above are satisfied). This record is accessed by the function F1 and another function not shown in the figure (say F3). The module candidate comprises the selected record, the functions F1 and F3 (Step 3).

Please note that in general a module may comprise multiple records (e.g., if F3 accesses another record matching the heuristic rules). However, in the case of the AXE10 system each module is typically grouped around a single record.

What would have happened if we had neglected the domain knowledge on the realization of modules and if we had just used the Liu and Wilde algorithm as proposed? In that case, Step 1 of the algorithm would refer to all global data elements and any kind of routines. By consequence, the algorithm would determine two sets $P(\mathsf{Var1}) = \{2, 3, \mathsf{F1}, \mathsf{F2}\}$ and $P(\mathsf{Record}) = \{3, 4, \mathsf{F1}, \mathsf{F3}\}$. Because these two sets have a non-empty intersection, the corresponding module candidate would comprise the contents of both $P(\mathsf{Var1})$ and $P(\mathsf{Record})$. Thus, all elements of the sample graph would be assigned to a single module.

[2] If function recovery has not been performed, the algorithm will work anyway.

5 Architecture Recovery

5.1 Motivation

So far, we have been concerned with the creation of a structure graph from the source code and other auxiliary documents. The structure graph strongly depends on the syntax of the underlying programming language (PLEX). In addition, it contains conceptual abstractions such as functions and modules which do not occur as syntactic units in the PLEX language.

Architecture recovery creates an architectural description of the system under study from the structure graph. The underlying *architecture description language* (ADL) has to meet the following requirements:

- The ADL has to be independent from the programming language(s) in which the system is written. In particular, language independence is important when different parts of the system under study are written in different programming languages. In the case of AXE10, which has originally been written exclusively in PLEX, more and more parts are being added in C++, resulting in a hybrid system.
- The ADL must support domain-specific abstractions and must be accepted widely by telecommunication experts.

The second requirement rules out general-purpose ADLs to which the world of processes communicating by exchanging signals via protocols defined by state machines cannot be mapped adequately. On the other hand, there are still multiple candidate languages meeting the stated requirements. In particular, the domain experts at Ericsson have applied both *SDL* [12] and *ROOM* [10] to the modeling of telecommunication systems. In the end, it was decided to use ROOM as the ADL in the E-CARES project, primarily because it provides clean concepts for modeling components interacting via ports along which messages are sent and received.

However, ROOM was not a clear winner over SDL. Furthermore, the UML component model [13] constitutes another candidate ADL which could be considered in future work[3]. For these reasons, we were aware that the target language for architectural recovery may be changed later on.

Therefore, we decided to decompose structural analysis into two *phases*: program analysis (as presented in Section 4) and architectural recovery. Furthermore, the architecture is represented separately from the structure in an architecture graph (with mappings into the structure graph). This design provides *modularity* of the E-CARES reverse engineering environment. Thus, the mapping to ROOM described below could be replaced easily with a similar mapping to SDL or to the UML components.

5.2 Real-Time Object-Oriented Modeling Language

The *Real-Time Object-Oriented Modeling* language [10] was designed for modeling asynchronous, concurrent, and distributed real-time systems. ROOM is

[3] When architectural recovery was realized in the E-CARES project, the UML 2.0 component model was still in flux.

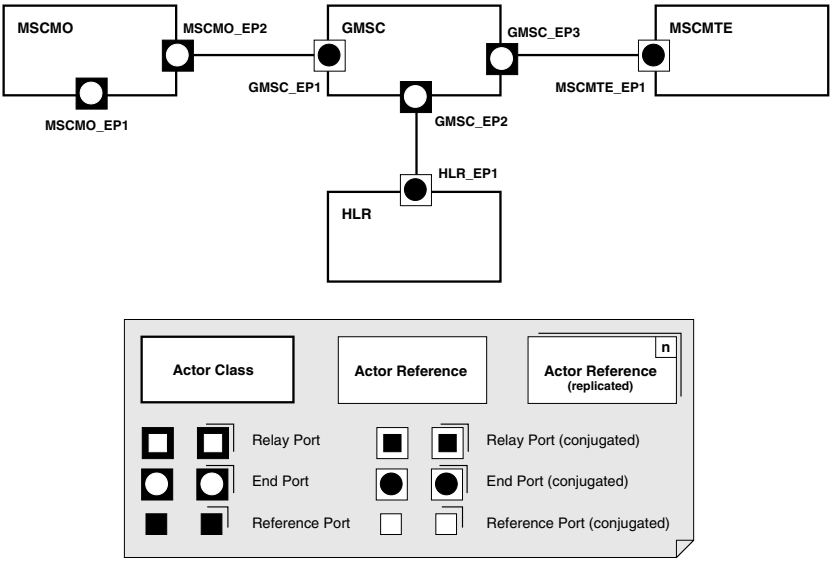


Fig. 7. ROOM diagram for the mobile originating call scenario (simple mapping)

the outcome of the authors' and their colleagues' combined practical experience in developing a variety of real-time systems in different industries including robotics, aerospace, and telecommunications. They claim that especially the development of a large distributed telecommunication system had a significant impact on the ROOM language.

A ROOM model consists of two parts: *structural models* and *behavioral models*. ROOM structure models are a kind of component inter-connection and component refinement diagrams. Behavior is modeled by means of extended state machines. Since this paper is concerned only with structural analysis, we will not discuss behavioral modeling below.

A simple example of a ROOM *structure diagram* is given in Figure 7[4]. Architectural components are represented by *actor classes*. Each actor class is displayed as a rectangle (e.g., MSCMO). The interfaces of actor classes are defined by *ports*, which are shown as small squares (with nested circles) placed on the border of the rectangle of the respective actor class. An actor communicates with its environment by sending and receiving *messages* via these ports. For each port, a *protocol* defines its incoming and outgoing messages. For some protocol p, its *conjugated protocol* is constructed by inverting the direction of messages. Finally, ports are connected by *bindings*. The ports connected by a binding must be *compatible*, the port at one end must be able to receive the messages sent via the port at the other end. This requirement is satisfied if the ports are associated to conjugated protocols.

[4] The legend refers to Figure 8, as well.

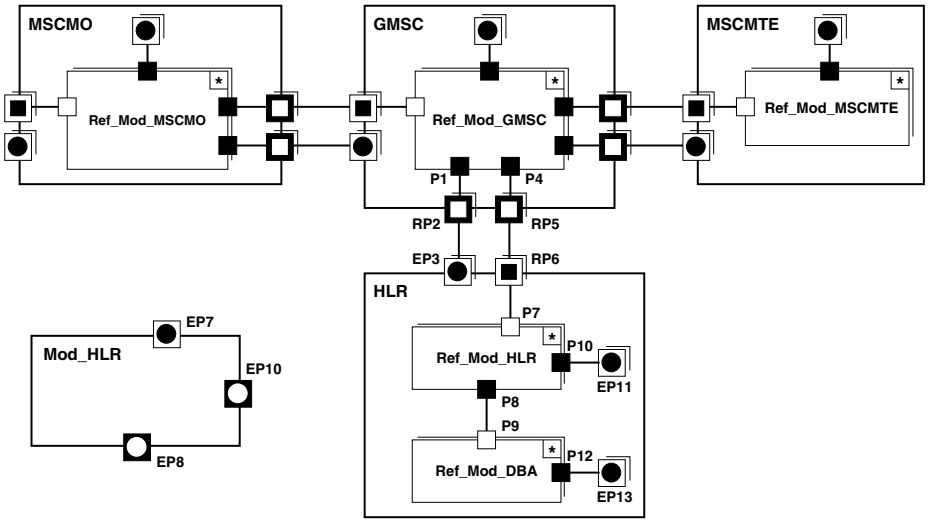


Fig. 8. ROOM diagram for the mobile originating call scenario (improved mapping)

Figure 8 shows a more complex structure diagram which makes use of further language constructs provided by ROOM. Actor classes may be refined into a subdiagram, which is shown within the rectangle for the actor class. Nodes of a subdiagram stand for *actor references*, i.e., references to instances of actor classes. In general, actor classes may be reused in different parts of the overall architecture; their instances are plugged into their respective context by connecting ports with bindings. In the case of double-shaped actor references, multiple instances may be created at run time (*replicated references*). The multiplicity is given in the upper right corner of the rectangle; * denotes an unbounded number of instances.

Furthermore, the diagram illustrates some generalizations and refinements concerning the modeling of ports. Analogously to replicated actor references, *replicated ports* are shown as double-shaped squares. In addition to *external ports*, an actor class may have *internal ports* which do not belong to its interface and therefore are shown inside (but near the border of) the respective rectangle. Furthermore, a distinction is made between class end ports and relay ports. *Class end ports*, which were shown already in Figure 7, are processed by the actor class itself; they are represented by squares with nested circles. In contrast, *relay ports* (nested squares) are used to pass messages upwards or downwards in the hierarchy without processing.

So far, we have introduced merely those elements of the ROOM language which are needed in the context of this paper. Further important language constructs such as inheritance on actor classes and layers are not exploited by the mapping algorithms below. *Inheritance* cannot be applied in architecture recovery since telecommunication systems programmed in PLEX are object-based

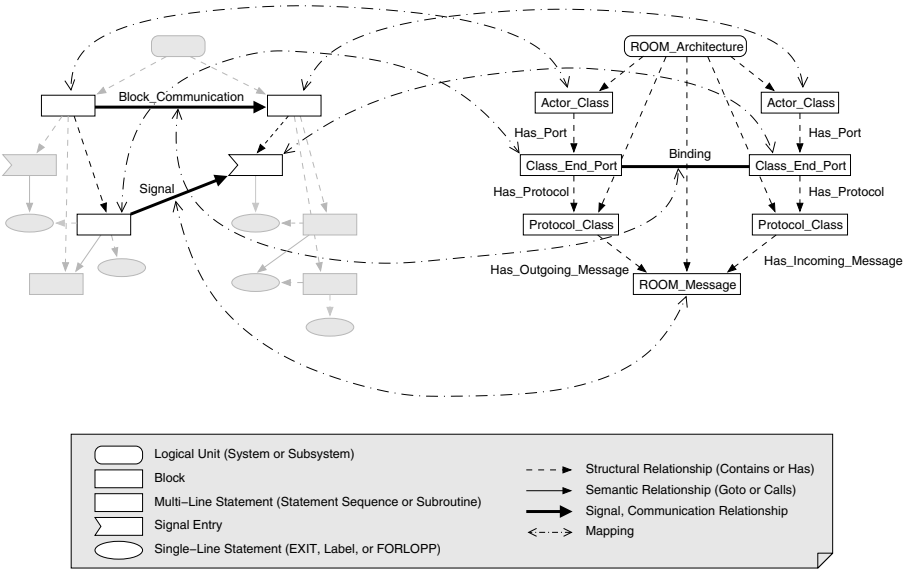


Fig. 9. Example mapping of a simple PLEX structure graph to a ROOM graph

rather than object-oriented. *Layers* are an important concept in telecommunication architectures. Unfortunately, no machine processable sources are available from which the layering of the AXE10 system could be inferred.

5.3 A Simple Mapping Algorithm

In general, the basic idea of the E-CARES approach to architecture recovery is to identify elements in the structure graph that can be considered to correspond to elements of the ROOM language. More precisely, this means that we have to identify elements in the PLEX structure graph that can be thought of as realizations of actors, ports, protocols, messages, bindings, etc. The identified structure graph elements are then mapped to newly created elements of an architecture graph.

First, we have defined a simple approach for mapping elements of the structure graph to ROOM elements (Figure 9) as follows:

1. PLEX blocks are mapped onto actor classes. Blocks are the active entities of a PLEX software system while actor classes are the active elements of a ROOM model.
2. Each block communication is mapped onto a corresponding binding. Before creating the binding, the ports to be connected have to be created. Thus, ports bundle all messages sent to or received from one destination.

3. Signals are mapped onto messages.
4. The protocol class attached to a port is derived from signal entries and signal sending statements, i.e., from the fine-grained communications which are aggregated into a block communication.

With respect to the running example of a mobile originating call and the participating blocks MSCMO, MSCMTE, GMSC, and HLR (see Figures 2 and 3), the architecture diagram resulting from this simple mapping algorithm is depicted in Figure 7. The diagram shows some characteristics typical for the type of ROOM structure diagrams produced by the simple mapping algorithm. Obviously, the diagrams consists of actor classes only. Furthermore, there is always a single communication contract between a pair of communicating actor classes. The ports bound to one of these communication contracts have conjugated protocols in the sense that no port is able to receive messages which cannot be produced by the other port. Finally, the port MSCMO_EP1 symbolizes that there might be ports that have no connection to any communication path. This may indicate dead code (received signals which are not sent) or may result from a constrained scope of program analysis (confined e.g. to some subsystem).

Comparing the results of this first and simple mapping algorithm with the information already provided by the structure graph, the improvement is quite small. The simple algorithm can be considered an *unparsing algorithm* that converts the block communication level of the structure graph into a standardized notation. The only improvement is that interfaces are now clearly visible in terms of ports and that the signals sent and received by a block are associated to different protocols that can be named, described and compared easily.

5.4 An Improved Mapping Algorithm

So far, we have merely performed a syntactic transformation from PLEX into ROOM. In the following, we will present a more elaborate algorithm which exploits *methods of use*. Once again, this mapping demonstrates the importance of domain knowledge beyond the language level.

The simple mapping algorithm presented so far does not consider any decomposition of blocks and actor classes, respectively. But, as already elaborated, most blocks in a PLEX software system contain at least one *module* (which is a conceptual rather than a syntactical unit). In addition, there can be several *instances* of a block or, more precisely, of its modules being the active entities of a PLEX system at runtime (see Subsection 2.3). Furthermore, we have already mentioned that the instances of a block are created, managed, and destroyed by the block itself.

Consequently, the actor classes resulting from architecture recovery should show a decomposition into several actors representing the module objects contained in the corresponding block. In addition, there should be a possibility to access the actors in the decomposition frame of an actor class from its behavior component that represents the control interface for instance management. Taking these considerations into account, our mobile originating call scenario is mapped into the refined structure diagram shown in Figure 8.

The improved mapping algorithm proceeds as follows:

1. Each block is mapped onto a complex actor class. In our example, the actor classes named MSCMO, GMSC, MSCMTE, and HLR are created.
2. Each module is mapped onto an an atomic actor class. In Figure 8, only the actor class Mod_HLR is shown as an example in the lower left corner.
3. Since as many module instances as needed are created at runtime, replicated references with multiplicity * are placed inside in the actor class for the surrounding block. In the example, the actor classes MSCMO, GMSC, and MSCMTE contain references to only one nested actor class because each of the corresponding blocks contains only one module. Since the block HLR contains two modules (one implementing the application logic of the block and one for the database), the corresponding actor class contains references to two nested actor classes.
4. For each block, a class end port is added to the interface of its corresponding actor class. This port represents the management interface of the block. In particular, the management interface handles SEIZE signals requesting the creation of link chains.
5. Similarly, a class end port is created for each actor class representing a module. For example, EP10 represents the interface which is in charge of managing the instances of module Mod_HLR.
6. For connecting to the management ports of nested modules, corresponding internal class end ports are generated within the respective actor classes (e.g., port EP11 in HLR).
7. For each pair of communicating modules (locally or across different blocks), class end ports are created on either side. For example, the port EP7 of actor class Mod_HLR is created for the connection to Mod_GMSC, and EP8 is going to be connected to a corresponding port of Mod_DBA.
8. For local communication among modules of the same block, bindings are created among the ports of actor class references. In this way, the binding between ports P8 and P9 within the actor class HLR is created.
9. If communication is performed between two modules belonging to different blocks, bindings cannot connect references to actor classes for modules directly because direct connections would break encapsulation. Rather, communication is routed via relay ports attached to the actor classes of the respective surrounding blocks. For example, for the communication between Mod_GMSC and Mod_HLR the relay ports RP5 and RP6 are created and connected with each other. Furthermore, they are connected to the ports of the respective actor class references acting as source or sink of the communication (ports P4 and P7, respectively).

Let us briefly sketch how a link chain would be set up at run time: First, the block MSCMO receives a request for setting up a link chain via its management port (lower port on the left-hand edge of the rectangle). Through its internal class end port (close to the upper edge), it delegates the request to its nested module Mod_MSCMO via its management port (located on the upper edge of the rectangle). In response to this request, an instance of this module is created.

To embed this instance, instances of relay ports of the surrounding block are created and connected to the respective ports of the module instance. Via the lower port shown on the right-hand edge, a message is request is sent to the next block in the link chain (GMSC). The reply returns a reference to a new instance of Mod_GMSC, and subsequent communication with this instance is performed via the new instances of the upper relay ports on the left-hand side of MSCMO and the right-hand side of GMSC, respectively.

Altogether, the improved algorithm constructs an architecture diagram which makes heavy use of domain-specific *design patterns*. In this way, the architecture diagram reflects the methods of use employed as design guidelines at Ericsson.

6 Realization

An overview of the E-CARES environment has been given in Section 3. To reduce the effort of implementing the E-CARES environment, we made extensive use of generators and reusable frameworks [17]. Scanners and parsers are generated with the help of JLex and jay, respectively. Graph algorithms are realized in PROGRES [18], a specification language based on programmed graph transformations. From the specification, code is generated which constitutes the application logic of the E-CARES environment. The user interface is implemented with the help of UPGRADE [19], a framework for building interactive tools for visual languages. Project specific extensions to the framework have been realized in Java.

A core part of the E-CARES environment consists of a (large and complex) PROGRES specification of structural and behavioral analysis. In this paper, we have given an informal presentation of structural analysis in E-CARES, mainly focusing on the construction of domain-specific abstractions for telecommunication systems. Due to the lack of space, we cannot elaborate in depth on the PROGRES specification for E-CARES. Rather, the reader is referred to [8,17,20] for detailed information in this area.

In the sequel, we present a small example from *program analysis* which demonstrates the application of PROGRES in the E-CARES environment. The graph transformation rule shown in Figure 10 is used to connect signal sending statements to matching signal entries. This rule is invoked after isolated structure graphs have been built by parsing PLEX blocks separately. For example, the application of this rule is used to generate the connection labeled SignalIAM in the structure graph of Figure 5.

Please note that Figure 5 shows a user-friendly, simplified representation of the structure graph. In particular, the connection SignalIAM is represented internally by a node and two adjacent edges rather than by a single edge. This way of representation is necessary because the underlying data model (attributed graphs) does not allow to attach attributes to edges (which is required in the case of the connection to be created).

The graph transformation rule is supplied with a parameter which determines the node for the signal sending statement to be connected (signalNode). Due to

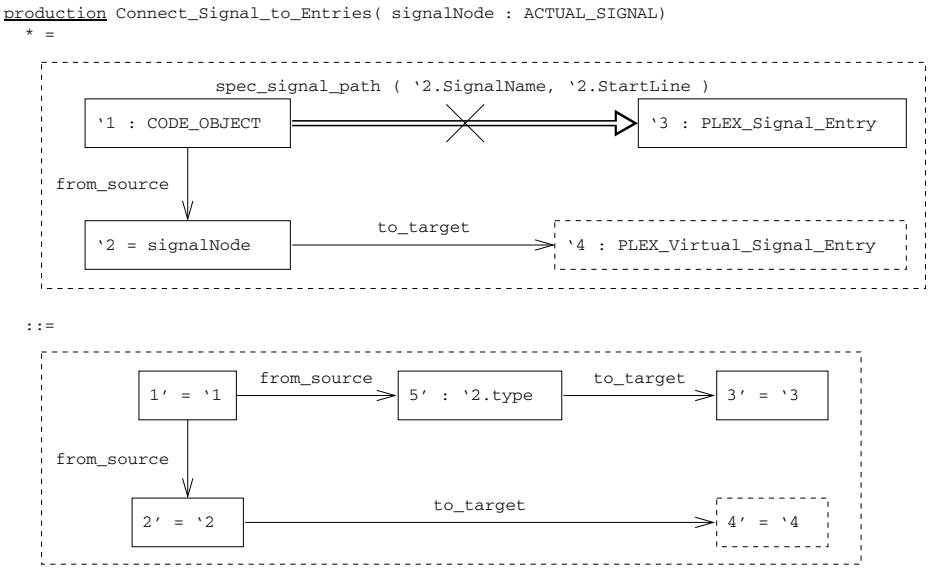


Fig. 10. Graph transformation rule for connecting sent signals to signal entries

the operator * (second line), the rule is applied to all matching subgraphs rather than just once. This is essential for handling multiple signals, which have to be connected to all potential recipients.

On the left-hand side (upper dashed box), which specifies the subgraph to be replaced, node '2 is fixed by the input parameter signalNode. When processing a single block without context information, the signal sending is connected to a virtual signal entry ('4), which acts as a placeholder for the final receiver(s). Both nodes are removed later in a clean-up phase; therefore, they are not contained in the sample graph of Figure 5. Node '1 represents the surrounding code object to which the signal sending statement will be lifted (statement sequence DIAL001_cs in Figure 5). Node '3 represents the signal entry to which the signal sending statement will be connected. Candidate nodes are retrieved by name matching (see the last line in the condition part which compares names attributes). In addition, a connection must not have been established yet, i.e., the (crossed out) path from '1 to '3 must not exist.

The right-hand side (lower dashed box) describes the transformation to be executed when a match of the left-hand side has been found (satisfying also the condition part of the rule). Here, the node 5' and two adjacent edges are created

(corresponding to the connection SignalIAM in Figure 5). The transfer part assigns values to various attributes attached to the node (SignalName, Comment, ...). All other parts of the graph remain unaffected.

It is interesting to note that methods of use have been taken into account in the specification of this graph transformation rule. As already mentioned at the end of Subsection 4.1, telecommunication experts at Ericsson employ the *phase division idiom* if processing of a signal requires more than a single time slice. Thus, CONTINUE signals have to be ignored when connecting single sending statements to single entries. For this reason, the condition part of the graph transformation rule requires that the name of the signal must not be equal to CONTINUEB or CONTINUEC (more than three cycles are never needed).

Architecture recovery does not augment the structure graph. Rather, it builds a new representation (an architecture graph). This allows for a clean separation of program analysis and architecture recovery, and it also permits multiple alternative architectural representations in different ADLs. Technically, architecture recovery is realized by a *triple graph transformation system* which is based on *triple graph grammars* [21,22]. A triple graph grammar is composed of rules operating on three graphs – a source graph (the structure graph), a target graph (the architecture graph) and a correspondence graph storing the mappings between the elements of the source and the target graph. In the original triple graph grammar approach, synchronous rules (extending all graphs simultaneously) are specified first, and directed rules (e.g., source-to-target transformations) are generated from synchronous rules. In our work, we specified directed transformations from the structure graph into the architecture graph immediately without coding the synchronous rules first (only one direction of transformation was needed in the E-CARES reverse engineering environment). For further details, the reader is referred to [8].

7 Related Work

The E-CARES prototype is a reengineering environment designed for telecommunication systems. In particular, it is based on domain-specific architectural concepts. A telecommunication system is modeled as a set of active components which communicate by sending and receiving signals. Thus, modeling is process-centered. Since the static system structure is still strongly dependent on the syntax of the underlying programming language (PLEX), additional conceptual abstractions such as functions and modules have been added. From the structure graph, a programming language independent architectural description of the system under study is created. As telecommunication systems are designed in terms of layers, planes, services, protocols, etc., it has been crucial to choose an ADL which supports domain specific abstractions.

While E-CARES also considers the behavior of the system under study [14], many other reengineering tools such as e.g. Rigi [23] or KOGGE [24] primarily focus on the static system structure. Moreover, they are typically data-centered; consider e.g. tools for the reengineering of COBOL programs as described in

[3,4,25]. Here, recovery of units of data abstraction and migration to an object-oriented software architecture play a crucial role [26]. More recently, reengineering has also been studied for object-oriented programming languages such as C++ and Java. E.g., TogetherJ or Fujaba [27] generate UML class diagrams from source code.

The phases extraction, post-processing, and inspection can be found in a similar form in different reverse or reengineering approaches. A typical example is the *extract-abstract-view* metaphor described in [28], which consists of three phases that occur in terms of activities in our process, too. In this abstract form, the metaphor also serves as a kind of reference architecture for reverse engineering tools. In particular, integrated reverse engineering tools like DALI [29], GUPRO [30], PBS [31], ReForDi [3], Rigi [23,32], Bauhaus [33], ANAL/SoftSpec [28], and SWAGKit [34] follow this reference architecture. The E-CARES reengineering environment does so, too.

Architecture recovery means to bridge the gap between implementation and architecture specification using appropriate reverse engineering techniques and tools. Basically all of these approaches employ some kind of information retrieval facility that extracts information artifacts from the subject system's source code. The techniques used there range from parsers [33] to string pattern matchers like the UNIX-tool grep or ESPaRT [35,36].

In general, we identified two different classes of approaches to solve this problem – approaches using any kind of domain information, e. g. in terms of patterns, and approaches employing more general-purpose complex flow analysis techniques. For example, Fiutem et al extract information on systems, programs or modules by means of architectural recognizers [37] working on the Abstract Syntax Tree (AST).

Another architecture reconstruction method that employs patterns to identify architectural components and connectors is described by Guo et al [38]. Their ARM method first obtains knowledge on the as-designed architecture. Then, this knowledge is used to define queries for potential patterns which are applied automatically to extracted and fused source model views. ARM uses lower-level patterns to build higher-level patterns and supports composite patterns as well. Similar work has been described in [36]. In contrast to the former approach, this approach by Pinzger and Gall also considers the knowledge of systems experts in pattern definition.

If the formulation of suitable patterns is impossible for any reason, approaches using elaborate control and data flow analysis are more suited and produce better results. Such approaches recover architectural components and connectors based on other criteria like *low coupling* and *strong cohesion* [15,16,39]. In this context, strongly cohesive code artifacts are considered to belong to the same component while the relationships between the different groups of cohesive artifacts become connectors of the resulting architecture. A possible technique to identify architectural components is, for example, program slicing and concept lattices as described in [33].

Comparing these approaches with the E-CARES approach, a significant difference to the majority of approaches is that we clearly separate architecture recovery specific activities from activities generally performed in structural or behavioral analysis. To the best of our knowledge, our approach is the only one that considers domain specific modeling languages and notations in architecture recovery and thus provides a flexible approach that allows to consider *different* of such architectural styles in the same tool environment. The separation of general structural and behavioral analysis from architecture recovery allows to share major parts of the existing functionality in different recoveries.

The E-CARES approach does incorporate patterns on different levels as well as control and data flow analysis. In structural analysis, the patterns in E-CARES are domain-specific *code patterns* in most cases rather than high-level or general purpose design patterns. In architecture recovery, design patterns above code level have been used. An approach to introduce high-level connectors in E-CARES as defined by Herzberg can be found in [11]. Instead of utilizing design patterns, we currently combine knowledge on code patterns (idioms) with control-flow analysis and data flow analysis to create abstractions in the structure graph. Then, elements of the structure graph are mapped to corresponding elements of an architectural representation. We agree with the pattern-using community that existing knowledge obtained from experts and (design) documents is mandatory in architecture recovery — but also in program analysis.

8 Conclusion

We have presented the E-CARES approach for reengineering of telecommunication systems. In this paper, the focus has been on the reverse engineering tools of the E-CARES prototype, specifically on structural analysis (program analysis and architecture recovery). In addition to structural analysis, E-CARES supports behavioral analysis to enrich the results of structural analysis with additional information. These reverse engineering tools aid in system understanding but not yet in system restructuring. Recently, the E-CARES prototype has been extended to also cover re-design and code transformation [9]. Multi-language support as well as extensions to the ROOM architecture description language have been elaborated in subprojects.

Methods of use have played an important role in structural analysis. In program analysis, conceptual units such as labeled statement sequences, functions, and modules are recovered which do not correspond to syntactic units of the PLEX language. For example, modules group functions which access a common data record for managing an instance of a block (for serving a specific call). Furthermore, in architectural recovery ROOM diagrams are constructed which incorporate instances of domain-specific design patterns. For example, an actor class for a block is connected to its environment via two ports which handle messages at different levels of operation (ports for creating/deleting block instances and ports for communicating with specific instances, respectively). Altogether, domain-specific knowledge has proved indispensable in constructing domain-specific abstractions which are meaningful to telecommunication experts.

Thus, all tools in E-CARES were developed in close cooperation with telecommunication experts from Ericsson. We followed an evolutionary approach to tool development, i.e., functionality was added incrementally in response to the requirements stated by the telecommunication experts. In this way, we took a step towards an environment which is based on domain-specific concepts. Though domain knowledge was widely used in extraction logic, the E-CARES prototype is a modular reengineering system such that major parts can be reused when analysing software from other domains.

To date, approximately 2.5 million lines of PLEX code plus several ten thousands of lines of additional documents have successfully been processed, analyzed, visualized, and inspected on different levels of abstraction. Understandably, the exact and detailed results are confidential and cannot be discussed here. According to the telecommunication experts, the E-CARES prototype allows to visualize the AXE10 software system in terms of their daily use, e.g., block dependencies, state diagrams, link chains and sequence diagrams. For that, no tools have been available so far. In particular, the dynamic analysis tool and the architecture extraction have proved their value for system analysis and system understanding. Therefore, we believe that only the combination of static and dynamic as well as structural and behavioral analysis – integrated in an interactive reengineering framework – allows to obtain best possible results.

Acknowledgments. The work reported in this paper was performed while both authors were members of the group of Manfred Nagl (Department of Computer Science 3, RWTH Aachen University). Manfred set up the E-CARES project in cooperation with Ericsson. Funding from Ericsson is gratefully acknowledged. Furthermore, we are indebted to all persons at Ericsson who provided support in different ways (Ari Peltonen, Dominikus Herzberg, Martin Blumbach, Jörg Bruß, Dietmar Wenninger, and Andreas Witzel).

References

1. Chikofsky, E.J., Cross II, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Software 7(1), 13–17 (1990)
2. Kazman, R., Woods, S.G., Carrière, J.: Requirements for integrating software architecture and reengineering models: CORUM II. In: Proceedings 5th Working Conference on Reverse Engineering (WCRE 1998), Hawai, USA, pp. 154–163. IEEE Computer Society Press, Los Alamitos (October 1998)
3. Cremer, K., Marburger, A., Westfechtel, B.: Graph-based tools for re-engineering. Journal of Software Maintenance and Evolution: Research and Practice 14, 257–292 (2002)
4. Markosian, L., Newcomb, P., Brand, R., Burson, S., Kitzmiller, T.: Using an enabling technology to reengineer legacy systems. Communications of the ACM 37(5), 58–70 (1994)
5. Kollmann, R., Selonen, P., Stroulia, E., Systä, T., Zündorf, A.: A study on the current state of the art in tool-supported UML-based static reverse engineering. In: van Deursen, A., Burd, E. (eds.) Proceedings 9th Working Conference on Reverse Engineering (WCRE 2002), Richmond, VA, pp. 22–32. IEEE Computer Society Press, Los Alamitos (November 2002)

6. Wilde, N., Scully, M.: Software reconnaissance: Mapping program features to code. Journal of Software Maintenance: Research and Practice 7(1), 49–62 (1995)
7. Marburger, A., Herzberg, D.: E-CARES research project: Understanding complex legacy telecommunication systems. In: Sousa, P., Ebert, J. (eds.) Proceedings 5th European Conference on Software Maintenance and Reengineering (CSMR 2001), Lisbon, Portugal, pp. 139–147. IEEE Computer Society Press, Los Alamitos (2001)
8. Marburger, A.: Reverse Engineering of Complex Legacy Telecommunication Systems. Informatik. Shaker Verlag, Aachen (2005)
9. Mosler, C.: Graphbasiertes Reengineering von Telekommunikationssystemen. Informatik. Shaker Verlag, Aachen (2009)
10. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Reading (1994)
11. Herzberg, D.: Modeling Telecommunication Systems: From Standards to System Architectures. PhD thesis, RWTH Aachen University, Aachen, Germany (2003)
12. Ellsberger, J., Hogrefe, D., Sarma, A.: SDL - Formal Object-Oriented Language for Communicating Systems. Prentice Hall, Upper Saddle River (1997)
13. Object Management Group Needham, Massachusetts: OMG Unified Modeling Language (OMG UML), Superstructure, V 2.1.2. formal/2007-11-02 edn. (November 2007)
14. Marburger, A., Westfechtel, B.: Tools for understanding the behavior of telecommunication systems. In: Proceedings 25th International Conference on Software Engineering (ICSE 2003), Portland, OR, pp. 430–441. IEEE Computer Society Press, Los Alamitos (2003)
15. Liu, S.S., Wilde, N.: Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery. In: Proceedings of the Conference on Software Maintenance (CSM 1990), San Diego, CA, pp. 266–271. IEEE Computer Society Press, Los Alamitos (1990)
16. Canfora, G., Cimitile, A., Munro, M.: An Improved Algorithm for Identifying Objects in Code. Software - Practice and Experience 26(1), 25–48 (1996)
17. Marburger, A., Westfechtel, B.: Graph-based reengineering of telecommunication systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 270–285. Springer, Heidelberg (2002)
18. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages, and Tools, vol. 2, pp. 487–550. World Scientific, Singapore (1999)
19. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: A framework for building graph-based interactive tools. In: Mens, T., Schürr, A., Taentzer, G. (eds.) Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2002), Barcelona, Spain. Electronic Notes of Theoretical Computer Science, vol. 72-2, pp. 149–159 (October 2002)
20. Marburger, A., Westfechtel, B.: Behavioral analysis of telecommunication systems by graph transformations. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 202–219. Springer, Heidelberg (2004)
21. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
22. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. In: Heckel, R. (ed.) Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques. Electronic Notes in Theoretical Computer Science, vol. 148, pp. 113–150. Elsevier Science, Amsterdam (2006)

23. Müller, H.A., Wong, K., Tilley, S.R.: Understanding Software Systems Using Reverse Engineering Technology. In: The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences ACFAS 1994, Montreal, Canada, pp. 41–48 (May 1994)
24. Ebert, J., Süttenbach, R., Uhe, I.: Meta-CASE in practice: A case for KOGGE. In: Olivé, À., Pastor, J.A. (eds.) CAiSE 1997. LNCS, vol. 1250, pp. 203–216. Springer, Heidelberg (1997)
25. van Zuylen, H.J. (ed.): The REDO Compendium: Reverse Engineering for Software Maintenance, p. 405. John Wiley & Sons, Chichester (1993)
26. Canfora, G., Cimitile, A., De Lucia, A., Di Lucca, G.A.: Decomposing Legacy Systems into Objects: An eclectic approach. Information Technology and Software 43(6), 401–412 (2001)
27. Zündorf, A.: Rigorous Object-Oriented Development. Habilitation thesis, University of Paderborn, Paderborn, Germany (2002)
28. Lange, C., Sneed, H.M., Winter, A.: Applying the graph-oriented GUPRO Approach in comparison to a Relational Database based Approach. In: Proceedings of the 9th International Workshop on Program Comprehension IWPC 2001, pp. 209–218. IEEE Computer Society Press, Los Alamitos (2001)
29. Kazman, R., Carrière, S.J.: Playing Detective: Reconstructing Software Architecture from Available Evidence. Journal of Automated Software Engineering 6(2), 107–138 (1999)
30. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO – Generic Understanding of Programs: An Overview. Electronic Notes in Theoretical Computer Science 72(2), 10 pages (2002), http://www.elsevier.nl/locate/entcs/volume72.html
31. Finnigan, P.J., Holt, R.C., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H.A., Mylopoulos, J., Perelgut, S.G., Stanley, M., Wong, K.: The Software Bookshelf. IBM Systems Journal 36(4), 564–593 (1997)
32. Müller, H.A., Orgun, M.A., Tilley, S.R., Uhl, J.S.: A Reverse Engineering Approach To Subsystem Structure Identification. Journal of Software Maintenance: Research and Practice 5(4), 181–204 (1993)
33. Koschke, R.: Atomic Architectural Component Recovery for Program Understanding and Evolution. Doctoral thesis, Institute of Computer Science. University of Stuttgart, Stuttgart, Germany, p. 414 (2000)
34. Holt, R.C., Malton, A., Davis, I., Bull, I., Trevors, A.: SWAGKit – The Software Architecture Toolkit. Software Architecture Group, University of Waterloo, Canada (2003), http://swag.uwaterloo.ca/swagkit/
35. Knor, R., Trausmuth, G., Weidl, J.: Reengineering C/C++ Source Code by Transforming State Machines. In: van der Linden, F.J. (ed.) Development and Evolution of Software Architectures for Product Families. LNCS, vol. 1429, pp. 97–105. Springer, Heidelberg (1998)
36. Pinzger, M., Gall, H.: Pattern-Supported Architecture Recovery. In: Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002), Paris, France, pp. 53–61 (June 2002)
37. Fiutem, R., Tonella, P., Antoniol, G., Merlo, E.: A Cliché-Based Environment to Support Architectural Reverse Engineering. In: Proceedings of the 1996 International Conference on Software Maintenance (ICSM 1996), Monterey, CA, USA, November 4-8, pp. 319–328. IEEE Computer Society Press, Los Alamitos (1996)

38. Guo, G.Y., Atlee, J.M., Kazman, R.: A Software Architecture Reconstruction Method. In: Donohoe, P. (ed.) Proceedings of the First Working IFIP Conference on Software Architecture (WICSA 1999), San Antonio, Texas, USA, February 22-24, pp. 15–34. Kluwer Academic Publishers, Dordrecht (1999)
39. Burd, E., Munro, M., Wezeman, C.: Analysing Large COBOL Programs: the extraction of reusable modules. In: Proceedings of the 1996 International Conference on Software Maintenance ICSM 1996, Monetery, Canada, pp. 238–243. IEEE Computer Society Press, Los Alamitos (November 1996)

Do We Really Know How to Support Processes? Considerations and Reconstruction

Stefan Jablonski

University of Bayreuth
Institute for Informatics
Stefan.Jablonski@uni-bayreuth.de

Abstract. Since approximately 20 years process management is regarded as innovative technology both for the description of complex applications and for supporting their execution. Soon after the beginning of this process era, an inflation of process management systems started to flood the market. However, many users were very frustrated when they had to experience that these process management systems did not support their applications properly. One of the main causes for failing process applications was that process management was totally underestimated. In this contribution we try to get to the bottom of process management, i.e. we reconstruct the real requirements of process management which are the basis for the construction of working process solutions.

Keywords: Process Management, Process Modeling, Process Execution, Workflow Management.

1 Introduction

Since approximately 20 years, process management is regarded as innovative technology both for the description of complex applications and for supporting their execution. Soon after the beginning of this process era, an inflation of process management systems started to flood the market. However, many users were very frustrated when they had to experience that these process management systems did not support their applications properly.

One of the main causes for failing process applications was that process management was totally underestimated. Defining processes is more than painting "bubbles-and-arcs" pictures. Developing processes requires both a profound methodology and a powerful and expressive process modeling language. Executing processes is more than executing steps in a fixed and predefined order like jobs in a batch queue. Supporting process execution requires an infrastructure that definitely reflects mandatory restrictions imposed by applications but also sufficient flexibility must be provided such that users are not limited in their creativity.

The contribution of this paper is not a new method for process management; rather it intends to reveal fundamental issues of process management (process modeling and process execution). This investigation should inspire researchers to contemplate current developments and research areas; it should foster to develop new concepts for

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 393–410, 2010.

process management that are better coping with the requirements of process based applications. Especially, it will stimulate researchers to more thoroughly analyze requirements of process based applications. It will therefore not focus on the introduction of a new technical framework to deal with process management; rather, it wants to illustrate unconventional ways of process modeling and execution which foster flexible process management.

Section 2 identifies problems with process modeling and execution. Section 3 then introduces possible solutions to those problems. Section 4 closes the paper with a short conclusion and recommendation.

2 Requirements Analysis

As said before, process management is good for the support of complex application systems. We assume certain complexity since without that the usage of process management might be too much of a good thing. It is like with database systems: if just 50 to 100 data items have to be managed it might not to be recommended to deploy a database system. The effort and the cost that would have to be put into the development of a database application would not pay off for such a small application. The real benefit of database systems appears when a huge number of data items have to be managed in a very flexible way.

The comparison of process management with database management shows some interesting parallelisms. Therefore, we will analyze the area of process management by leaning on database management. Also, we will distinguish two issues when dealing with data and processes, respectively. First, things are defined; then, these things are used. Hence, two major tasks of database management are "data definition" and "data usage". The former means to define a database schema, the latter means to work with the data that are stored in a database system (inserting, modifying, deleting, and reading them). Analogous, we identify the two tasks "process definition" and "process usage". Process definition means to define process models that show how (process oriented) applications are structured. Process usage usually aims at the enactment of processes, i.e. processes have to be applied or used in applications. Often, process enactment is equal to "process execution", i.e. a system is (more or less) strictly executing a process step by step. However, we will discuss that this perspective is rather too narrow. To analyze both process modeling and process usage, we compare these tasks to the corresponding tasks in data management.

2.1 Process Modeling

First, the topics data definition and process definition, respectively, are discussed. There are many ways to define data structures for database applications. ER modeling is one of the most popular ones [15] and therefore we want to concentrate on it. Without going into the discussion of "unconventional" applications, the ER modeling method has proved its worth. This statement holds for "conventional" applications like in administration, production management, banking and insurance applications. We are well aware that ER approach shows limitation when "unconventional" applications like CAD or scientific applications (e.g. modeling a DNA) has to be enacted. Nevertheless, the comparison of conventional database applications is good enough to encounter requirements for process management applications.

What are the characteristics of conventional applications and why does ER modeling fulfill their requirements? Applications are characterized by data. It is necessary to define them first and then to define relationships between them. Data and relationships have to be characterized by certain individual features. All this is possible by applying the ER method.

The strength of ER modeling lies in its simplicity. There are two major concepts: entities and relationships. Entities describe data and relationships show how they are depending on each other. Due to these simple concepts almost any arbitrary data and relationships can be defined.

Due to the elementary character of the ER modeling concepts, almost any data structure can be defined. Most notably, there is no prescription how to model application data. For instance, it is not prescribed how to model a data structure for project management. For one application, there might be a project entity having 20 attributes; for another application, the project entity might have 30 attributes, etc. The relationship between projects and project managers might be n:1 in the first application and n:m in the second application. Nobody would buy a database system with a fixed project table, showing certain "burned-in" attributes and relationships. This would be like a "universal database scheme" for all applications which would not be acceptable of course. It might even look ridiculous to discuss this feature here. Nevertheless, this freedom with respect to modeling data structures is the key benefit of database systems. This observation becomes obvious when we look into the area of process modeling.

In the area of process modeling there is no consensus on what information to include into a process model, e.g. for the Travel Claim Reimbursement Process. (By the way, a process model is a complex data structure defining how process participants - e.g. process steps, documents, tools - are related to each other.) As a consequence, an all-embracing process modeling language does not exist. At a first glance, this is not a problem. It is comparable to the fact that database systems do not have predefined "burned-in" tables for projects etc. However, database systems offer elementary modeling primitives to construct individual, application specific tables for any application, i.e. also for project descriptions. Yet, this capability is not given for process management systems (i.e. for process modeling languages). These languages do not offer modeling primitives to define individual process models. Because they don't do it, it might again be interesting to have an all-embracing process modeling language by which any process model can be derived. Now, the lack of it becomes an issue indeed.

Consequently, the main question now is whether it is more appropriate

- to offer an all-embracing process modeling language or
- to offer elementary modeling primitives (by a process modeling language) for the definition of arbitrary process models?

Let us first pursue the issue of an all-embracing process modeling language. From a historical perspective there are approaches like CPM (Critical Path Method, [2]) or PERT (Program Evaluation and Review Technique, [1]) that are suggesting what to include into a process model. However, there are many more approaches and not all of them are compatible with each other. These days, the BPMN standard (Version 1.2 [5], Version 2.0 [6]) becomes very popular, although it is not applicable to arbitrary applications [19]. So, what to include into a standard process model?

All the proposals for a common process modeling language mentioned above are not "complete", i.e. there are features which are not supported. For example, the CPM method does not allow modeling the organizational perspective of a process, i.e. organizational policies which determine who should execute a process. Other methods do not support well the modeling of data or applications that are needed to describe a process step (e.g. BPMN, before Version 2.0).

In principle all these approaches assume a certain template for processes which comprises a concrete, pre-determined set of concepts. However, these templates are not "generally applicable" but are all developed from a certain perspective and for a certain application domain. Thus, they comprise certain process features, and neglect others. In contrast, different application domains normally require different modeling features. Two examples support this thesis.

- The product development process of a car manufacturer must be modeled. One major issue here is to exactly model persons that are responsible for the execution of process steps. In order to do so, powerful organizational relationships between responsible persons must be expressed. For example, it must be defined that a part release step must be executed by the project manager who is responsible for the design of the car the part is made for. This latter policy assignment is a rather tricky one.
- In a health care application it might be required to model the legal regulations for medical treatments. This means that corresponding process steps must be associated with those legal regulations, i.e. there must be a link attribute between process steps and legal regulations.

Obviously, the car manufacturer and the healthcare application request different process modeling languages. We do not want to criticize any of these approaches mentioned above (e.g. BPMN, CPM). They might be very valuable for the application domain they are stemming from and they are made for. However, we criticize when providers of those approaches claim to have "the" generally applicable process modeling template. This is definitely not the fact and it is not possible since not all features of all applications can be incorporated into such a process modeling language. Therefore, we see the approach of having a "one-fits-all" process modeling language as not achievable. Consequently, we see the adoption of the data modeling approach as much more eligible. This means that a set of elementary process modeling primitives should be defined. Those primitives can be applied by domain experts to construct their customized process models comprising all features a process must show from their perspective.

As a summary for the requirements analysis for process modeling we identify the following: The definition of process modeling primitives is required that can be used to construct domain specific process models. This approach works similar as the ER data modeling approach; there the data modeling primitives are sufficient to define individual domain specific data structures.

2.2 Process Execution

Second, the topics data usage and process usage, respectively, are discussed. Before data can be used the database has to be prepared for it, i.e. data definition must take

place and data structures must be defined. To simplify the discussion and concentrate on the essential, there are two principle usages of data: data retrieval and data modification. Data retrieval is very clear: data are read and presented. Data modification comprises inserting, modifying and deleting data.

Nevertheless, not in all databases data modification occurs according to the same pattern, i.e. different database applications need different kinds of data modification operations. We want to differentiate between "normal" databases and data warehouses – i.e. OLTP vs. OLAP databases [4]. In normal databases an update operation is one of the normal operations: one or many tuples have to be updated. In contrast, in data warehouses update operations are more crucial. Normally, data warehouses are "append-only" databases: new tuples are inserted; existing tuples are not changed. Only in cases of errors (e.g. wrong data are corrected or missing data are completed) tuples are updated. While there are no special follow-on operations necessary in databases, in data warehouses complex and complicated follow-on operations are required. Typically, pre-aggregated data marts have to be re-compiled. Since these are very cumbersome tasks optimizations should be applied [3].

A similar observation holds for data retrieval. While data are normally retrieved from base tables in databases, in data warehouses pre-aggregations (e.g. in form of data marts) are necessary to efficiently process the queries. Additionally, while normal databases applications are mostly working on base data, data warehouse applications work on aggregated data.

In summary, we see that the usage of data is quite different in databases. It is so diverse that even two different implementation concepts are necessary (normal databases vs. data warehouses). The same observation holds for process management. Not all kinds of usage are the same. We will discuss this observation in the following.

After having modeled processes they can be used. What does that mean? We shortly want to present the recent development of the process management area and want to show that this development led into a wrong direction.

It was at the beginning of the nineties. Process management became very popular, especially due to Hammer's and Champy's publication [13]. In reaction to that, a huge number of process execution systems evolved. They were mostly called workflow management systems. Their common characteristic is that processes are strictly executed according to their definition, i.e. according to the underlying process model. What does that mean? Process models describe a certain order of process steps. A workflow management system then takes the process definition and interprets it strictly, i.e. one step after the other is executed. Experiences with that way of process execution were very disappointing. Mostly it was criticized that workflow management systems are too inflexible and therefore not applicable. We can follow this argument since we are convinced that this kind of process execution is much too restrictive. Nevertheless, it might be absolutely adequate in certain application scenarios. In order to argue that the strict execution semantics of workflow management systems is not generally applicable we illustrate how processes are executed in real life.

For example, we look into an administrational application. In order to prepare a meeting, a manager's secretary is writing its agenda and an invitation letter which includes the list of people who should be invited to that meeting. This list strongly depends on the topics on the agenda. The invitation letter has to be signed by the

manager. The invitation letter should not be signed by the manager before both the agenda and the list of participants is complete since he is responsible for them. In principle the corresponding process looks as shown in Fig. 1:

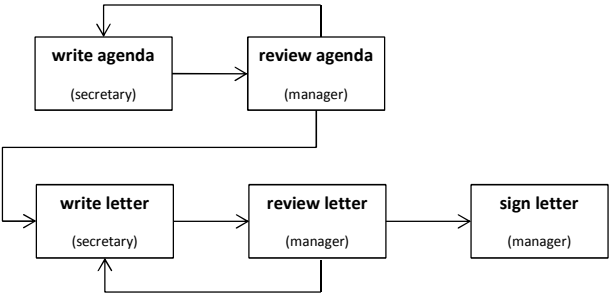


Fig. 1. Sample process

The process model in Fig. 1 seems to be correct. However, it is not practicable at all. In real life it will not be executed as shown in the figure since that would be too restrictive. The following situations might happen:

- After the manager has reviewed positively the letter or the agenda, one of these has to be changed again;
- After the manager has signed the letter, it must be corrected.

These are only two out of many "exceptional" executions that might happen. The question arises: why have those execution flows not been included into the process model before. We want to mention two reasons for that:

- The two above mentioned "exceptional" flows through the process are not the only ones. More are conceivable. To model all of them would increase drastically the complexity of the process model. Therefore, they are neglected;
- The two above mentioned "exceptional" flows are not modeled since they are not preferred and therefore they should not be offered.

Interpreting all of these arguments encounters a dilemma: on the one side, it cannot be avoided that "exceptional" executions like the above mentioned are relevant. If they are not modeled the process is much too restrictive and will most probably not be accepted by the users. On the other side, those executions should not be modeled since they are not recommended. Besides, to model all of them would enormously blow up the process model which then is not readable any more. How to resolve this dilemma?

Some consequences or requirements can be derived from the above observation:

- It is necessary to model "recommended" and "not recommended but nevertheless possible" paths in a process model. This is the only way to offer execution flexibility;

- In order to model these different kinds of execution paths, new modeling constructs are necessary. Just to use the conventional ones would make a process model unreadable.

As conventional modeling constructs we regard sequence, alternative and parallel executions and loops. These are the constructs which are borrowed from programming languages. Advanced modeling constructs also are able to model – among other things – recommended and exceptional paths.

To introduce new process modeling constructs defines new tasks for both process modeling and process execution. Therefore, we discuss this issue both in Section 3.2 and Section 3.3, respectively.

Another consequence of the scenario above should be considered that cannot be directly derived from the example. Why to restrict process execution to the way workflow management systems are doing it? Are there alternative ways of execution, i.e. usage of a process model? We think so. In Section 3.3 we will present such alternative interpretations.

3 Supporting Domain Specific Process Modeling and Execution

In Section 2.1 a few requirements for process modeling are identified. First, we look for modeling primitives. These modeling primitives can be used to define customized modeling constructs. Additionally, in Section 2.2 the need for new modeling constructs with new semantics has been identified. They are needed to express execution semantics beyond the usual execution semantics of workflow management systems. In this section we demonstrate how all these requirements can be met by a multi layer meta modeling approach. Presenting this approach does not mean that it is the optimal one; it just acts as a proof of concept. In the cited papers related approaches are discussed; this discussion should not be repeated here.

3.1 Process Modeling Primitives

At first, the issue of identifying modeling primitives is tackled. Therefore, we present the POPM (Perspective Oriented Process Modeling) approach; it is capable of offering process modeling primitives.

The POPM approach was first presented about 15 years ago in [16] and [17]; in [18] its conceptual backbone is reconstructed in detail. We refer to this publication throughout the following discussion.

The fundamental idea of POPM is not to offer process modeling constructs directly, but to offer a method for defining process modeling constructs. The enactment of the POPM method relies on a so called meta model stack. The basic idea of a meta model stack is the following: on level x (abstract) modeling elements are offered that can be used at level x-1 to define (concrete) modeling constructs. This approach resembles the rationale of MOF (Meta Object Facility, [5]). However, our approach heavily leverages on advanced modeling concepts like powertypes, deep instantiation, materialization, and clabjects which are not supported by the MOF approach (see [18] for details). Fig. 2 presents the POPM approach at a glance:

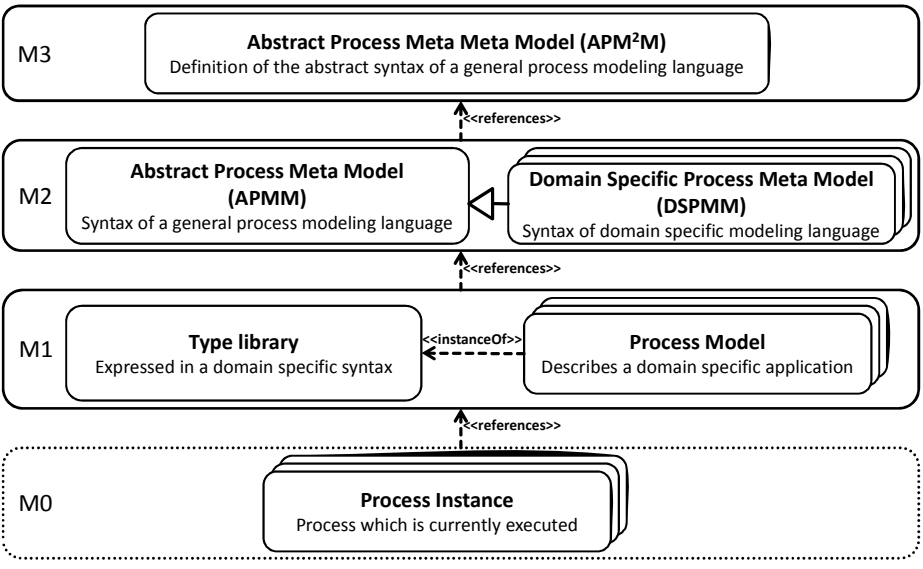


Fig. 2. Meta layer stack of POPM

Process models are defined on the modeling layer M1 (right boxes). A process model uses process (and data, organization etc.) modeling constructs which are accumulated in the "Type library" on M1 (left box). These modeling constructs are provided domain specifically; they are defined on the next upper level M2.

All process definitions on M1 are defined using the process modeling constructs offered by a process modeling language which is specified at level M2. Such a process modeling language is defined as a so called Domain Specific Process Meta Model (DSPMM). M2 further contains the definition of an abstract process meta model (APMM) defining a set of general language features for processes, data flow or control flow. These features are relevant for almost all process modeling languages; therefore they are collected in the APMM. All modeling elements on level M2 are process modeling primitives: They can be combined arbitrarily to set up domain specific process modeling languages. For instance, process modeling languages for the two examples in Section 2.1 (car manufacturer, health care application) can be defined. Each DSPMM defines an individual (we often say: domain specific) process modeling language (DSL: domain specific language). Thus, the above mentioned requirement is fulfilled.

Fig. 2 shows two further levels: M3 and M0. Level M3 offers generic modeling elements (boxes, arcs, etc.) which are used to define modeling primitives on level M2. Further, level M0 contains running instances of processes defined on M1 (right boxes).

It is not in the scope of this paper to prove that the modeling constructs offered on level M2 allow defining arbitrary process modeling constructs. We rather want to refer to [10] and [11] that describe various applications which all require specific process modeling languages; all of them could be provided by the POPM approach.

As a first resume, we can state that it is possible to offer individual process modeling constructs as requested in Section 2.1. Therefore, the lack of a one-fits-all standard process modeling language is not so critical. Especially, it does not prevent process modeling to be a broadly applicable method.

3.2 Domain Specific Process Modeling Constructs

Since the POPM approach supports the definition of new modeling constructs (assembled out of modeling primitives offered by layer M2) it is possible to define process modeling constructs with customized, individual semantics. This feature can be utilized to model compact modeling constructs or modeling constructs with specific execution semantics. These modeling constructs can be used, among other things, to support scenarios as presented in Section 2.2 (recommended execution paths). Nevertheless, just to offer new modeling constructs is not sufficient. Also new execution semantics must be provided. This will be done in Section 3.3.

Why introduce new process modeling constructs? There are principally two reasons:

- To represent certain scenarios with conventional modeling constructs would lead to unreadable and incomprehensible process models. Thus, new (e.g. compact) process modeling constructs are defined that model complex situations with plain means (i.e. the resulting process models are easy to read);
- In some scenarios specific execution semantics are necessary. To express this, new modeling constructs are needed.

Again as a proof of concept we introduce a piece of research work that successfully is coping with the two issues above. It is introduced in [19] and is called ESProNa. The ESProNa approach aims at process based applications with huge variability. Again, this is not to say that this research is best, it is just to present a successful feasibility study. Related approaches are discussed in the cited literature.

In [19] three new modeling constructs are introduced: two of them, namely two kinds of arrows will be presented in the following. The challenge for ESProNa is to present process applications as presented in Fig. 1 in an easy, readable way. Such process applications are characterized by showing a huge number of alternative execution paths. Although all of them should be offered, some of them should explicitly be marked as recommended; others should be marked as exceptional.

The following discussion only focuses on the control flow perspective of a process model, which this is absolutely sufficient for the purpose of this paper. ESProNa basically introduces two special process modeling constructs for control flow. The first process modeling construct is the normal arrow. The semantic of the well known arrow symbol in process modeling is that if an arrow directs from process A to process B then process B has to be performed after process A. Accordingly, if process B is connected with an arrow to process C then C may start after process B has finished. We also say: B requires the execution of A before it can run; C requires the execution of B (and consequently of A, too) before it can run. This arrow is represented by solid lines since it strictly prescribes an execution order (Fig. 3).

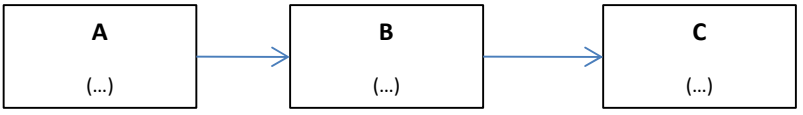


Fig. 3. The solid arrow

Beside this arrow construct depicted by a solid line we want to add an arrow depicted by a dashed line; this dashed arrow holds a different meaning. Two processes that are connected through a dashed arrow can – in principle – be executed in any order. For instance, if process A and process B are connected by a dashed arrow A can be performed before B or vice versa, B can be performed before A. Nevertheless, having defined a dashed arrow from process A to process B expresses a preference (recommendation) that process A should be performed before process B. This feature can be utilized when processes are put on a work list for execution. If more than two processes are connected through a dashed line then a permutation of all process executions is feasible, e.g. ABC, BCA, CBA (Fig. 4).

The introduction of the dashed arrow is a fundamental step with respect to the expressiveness of a process model. The interpretation of solid arrows is borrowed from classical logics [24]. There the "tertium non datur" holds; this is the principle of the excluded third. That means that if a proposition S holds (it is true) the inverse proposition ¬S does not hold (it is false); there are no other states than these two. In contrast, the interpretation of the dashed arrows more stems from constructive logics [22] [23]. There, only the principle of the "exclusion of contradictions" holds which is less strict than the "tertium non datur". It means that everything holds but those things that are explicitly excluded; in other words, no proposition can be true and false at the same time.

Also this is just a short and informal discussion it already reveals that by applying concepts of the constructive logics a much wider spectrum of execution paths result from a process model. The step from classical to constructive logics could be compared with the step from algorithm based computing to interactive computing that was discussed by Wegener [21]. It is not that this research work is directly comparable with our ideas it is just to say that a similar kind of design rationale lies behind these two research approaches.

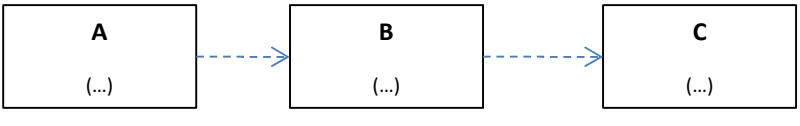


Fig. 4. The dashed arrow

It is certainly possible to combine the solid and dashed arrows. For example, process A and B are connected through a dashed arrow; process B and process C are connected through a solid arrow. This means that there is flexible ordering between processes A and B while process B must always be executed before process C. This semantics results in the following three execution orders: ABC, BAC and BCA (Fig. 5).

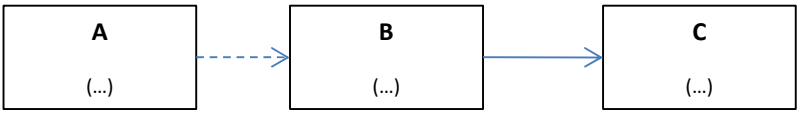


Fig. 5. Combination of dashed and strict arrow

Of course it is not enough that only new process modeling constructs are defined. Also an adequate execution infrastructure must be made available. ESProNa supports such an infrastructure. We will discuss this issue in Section 3.3.

Having available these two new process modeling constructs the example of Fig. 1 can be modeled as depicted in Fig. 6.

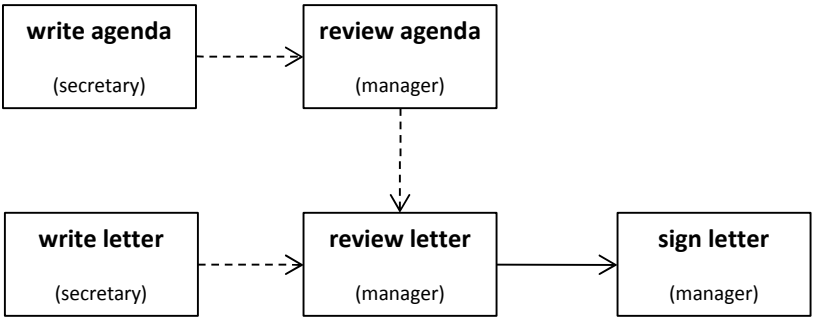


Fig. 6. The example of Fig. 1 – remodeled

At a first glance the revised process model does not look much simpler than the original process model from Fig. 1. However, we could argue that we have reduced the number of arrows from six to four. Although, this 33% reduction is a phenomena of the "statistics of small numbers" there is some truth with it. The process model from Fig. 6 is much more powerful than the process model from Fig. 1. Especially, all "exceptions" reported in Section 2.1 are covered by this revised process model. Additionally, the recommended path clearly appears in Fig. 6. The latter two arguments are the decisive ones. Besides, the process model is more readable. The attentive reader might have recognized that in the remodeled process also a review step can be performed before the corresponding write step is done. This is definitely intended: sometimes people want to bring forward a specific piece of work although they know that they cannot complete it. Nevertheless, they want to pre-executed the step since they want to adhere to a specific information in order not to forget it afterwards when they really complete the step.

In summary, we want to note that new, domain specific modeling constructs have the ability to model complex application scenarios with compact, well-arranged process models. Though, the process management community is requested to put more research into this issue since process models are the communication means between domain experts and (technical) process experts.

At the end of this section about domain specific process modeling we want to discuss alternative approaches. Of course, UML as the classical modeling language must be discussed. Since pure UML does not offer constructs that are sufficient for process modeling we look at extensibility mechanism of UML. The UML provides an extension mechanism that allows for creating Domain Specific Languages as well. Extensions are encapsulated within so-called Profiles which can be loaded into UML modeling tools [8][9]. These Profiles may not only specify language constructs but also a (graphical) notation for these constructs. However, with Profiles only such adaptations can be performed which do not alter or break the semantic of constructs contained within the UML. Therefore, language features that harm basic concepts of the UML such as Clabjects or Deep Instantiation cannot be put into a profile. Under the assumption that Clabjects and Deep Instantiation are fundamental for our approach, UML Profiles could not be taken to fully implement it.

3.3 Supporting Process Execution

In Section 2.2 the strict execution of processes through workflow management systems was criticized. Besides, it was motivated to think about alternative execution semantics. Thus, we want to go more into the direction of "process usage" than "process execution" – the latter always more or less directly directs to computer based enactment. Nevertheless, also computer based enactment strategies can provide flexibility that is needed by process users. In the following discussion we will stepwise "macerate" the strict execution semantics of workflow management systems.

We do regard the execution semantics as one very valuable interpretation of "process usage". There are application scenarios where this very strict execution strategy fits well. For example, when new construction drawings, CAD models, etc. have to be released within a design process, this has to be done according to fixed and strict rules. Responsible people have to release these artifacts, the latter have to be registered in corresponding repositories, and finally they have to be stored at predefined and predetermined locations. Each step within such a process must occur exactly in the right order. Thus, a workflow management system can perfectly support such a scenario.

However, it is obvious that the workflow management support is not suitable for other phases of a design process. For instance, in the more creative phases of a design process, no one wants to be strictly guided by a workflow management system. Rather, designers want to be free in their decision what to do next since design is a creative and non-predictable process. Nevertheless, designers would like to be disburdened by a system that helps them to find the right documents, tools, etc. when they are working on a specific step in the design process. For instance, when they want to do an FEM computation they want to be provided with all relevant documents (e.g. CAD model) and want to get offered the adequate tool for this computation. Also, they want to see whether all relevant documents are already available and might even want to get an overview about former occurrences of this design situation. Novice designers also are content when they are accommodated with an overview of the design process such that they get an idea what process steps to do next. Altogether, we call a system that supports all these mentioned issues a process navigator: it shows where a user stands in the process, depicts what resources (documents, tools, etc.) are

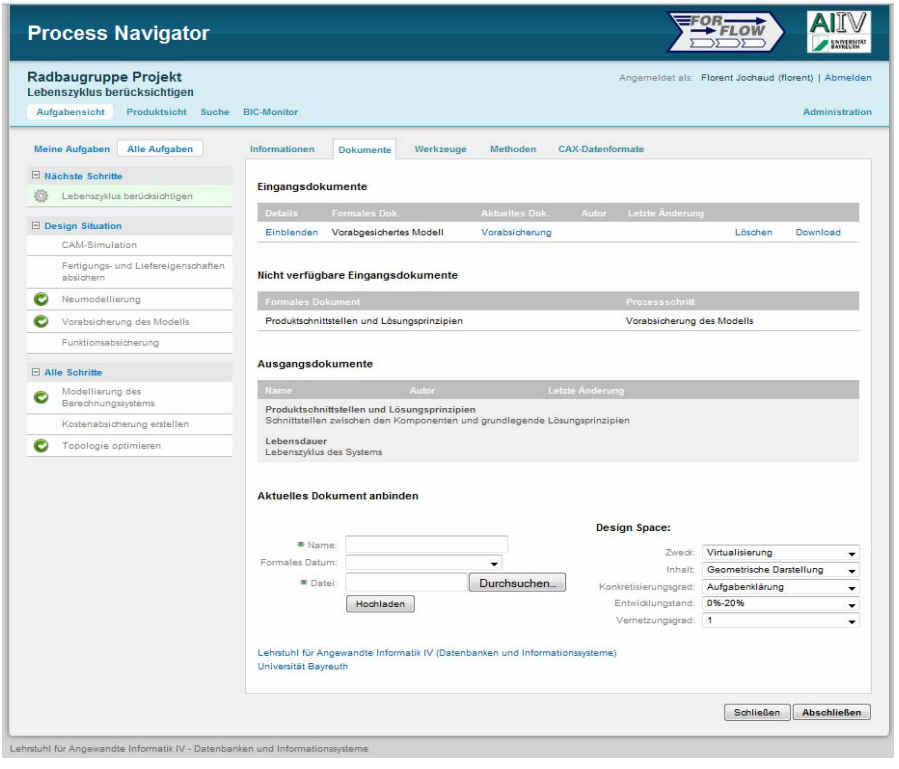


Fig. 7. The ProcessNavigator

currently available or needed, provides an outlook on what to do next and even allows browsing the history of former process executions. Again, as a proof of concept we will introduce a system that supports this special kind of process support: it is called ProcessNavigator [20]. We shortly present the ProcessNavigator in the following in terms of presenting a feasibility study, i.e. a systems that copes with the requirements posted so far.

The ProcessNavigator is depicted in Fig. 7. The left part of the ProcessNavigator provides different sorts of worklists. The upper worklist determines the process steps that – according to an underlying process model – should be executed next. However, the second worklists offers process steps that are – most probably – also relevant for the current situation. A third worklist acts as fall back and offers all process steps available in the process model. The user interprets the three worklists as follows: in the first worklists the highly recommended process steps are listed; the second worklist offers process steps that are fine but not really recommended. In order to cope with more exceptional situations the process steps from the third worklist can be selected.

Through the separation of different worklists the ProcessNavigator offers both guidance and flexibility for process execution. It is interesting to know that the ProcessNavigator is based upon the ESProNa method. Through its logic based implementation

not only recommendations about process steps can be given, also the ProcessNavigator can provide hints about effects of process executions. For example, if the process step "sign letter" is performed, no other process steps can further be executed. This functionality allows executing so-called what-if-games and again helping users to better navigate through a process.

The right part of the ProcessNavigator provides situation specific information. Among other things, it is depicted what documents are available for the execution of a specific process step, what tools could be applied, etc. Besides, historical data, i.e. data about former executions of the process can be retrieved. Altogether this information helps users to easier, faster and more consistently execute process steps.

Both the workflow execution mode and the ProcessNavigator execution mode are closely oriented to the underlying process model. Also, both execution modes take a process model and offer a user interface that allows a user to navigate – with more or less freedom – through a process model. In a project in the administration department of a university we found that alternative execution modes are also relevant. We call this new execution mode checklist method.

A second example once more stems from an administrational application. Again, a large process model is defined (> 250 process steps). Again, the variability of execution is huge. And what is more important, most of the process steps are so called manual tasks. We call a process step a manual task when it has to be worked upon mostly without computer support. For example, a paper document has to be signed and has to be sent to a following station. Nevertheless, the domain experts want to somehow be able to track and supervise process executions. Therefore, they developed the following concept.

First, all processes are modeled as usual. Then, for each process a so called checklist was defined. A checklist comprises the main process steps including documents that must be produced and agents that are responsible to perform the corresponding process. Agents had to sign it (electronically) when they have finished a certain process step. Through this method it was possible to track the execution of a process although most parts of the process were performed external to a computer system.

Main Process			
Process A	**Documents**: **IN**: Doc 1 **OUT**: Doc 2	**Comment**:	**Agent**: Y **Signature**: _____
Process B	**Documents**: **IN**: Doc 3 **IN**: Doc 4	**Comment**:	**Agent**: X **Signature**: _____

Fig. 8. Checklist

Fig. 8 depicts the principle structure of a sample checklist. It serializes the process steps of a process model and additionally shows what input sources could be used (Documents "IN"), what results are expected (Documents "OUT"), comments on the execution (Comment; that is very important to track experience), and who is eligible to perform the process steps (Agent). A signature – it can be either electronic or paper

based – confirms the execution of a process step. It has to be mentioned that the checklist method offers even more flexibility as the ProcessNavigator. In principle the one important statement is that at the end of the process all signatures are on the checklist. So it is completely output oriented. Nevertheless, the checklist method describes a very interesting and valuable form of process usage and widens its spectrum towards non-computer based and extremely flexible process execution.

Last but not least we want to introduce another form of process usage. It completely neglects computer support for process execution but only relies on process modeling. We motivate this approach with an example.

In a project with a sports and fashion company the production process for shoes has to be defined. The outcome of the modeling phase is a large process model comprising more than 200 process steps. Up to this point the project is absolutely conventional in the course of project management, i.e. conventional process modeling is encountered. However, the process "usage" is quite different to conventional process "execution". The company is recognizing that although the modeled production process is correct as a template, each concrete production process (for each production lot) is special and is deviating in many small places from the template process. Due to this variability it is not possible to model all these variants. Although there was no need to apply a tool like the ProcessNavigator since this kind of direct project control is not required. For the process is mainly used by the central headquarter department to supervise the process on an abstract level. Nevertheless, the company wants to make use of the production process model. Therefore, they choose the following kind of execution: The production process model is printed out whereby the process steps are arranged in a special way. Then this process model is stuck onto a wall as wall paper. With little colored flags the managers are indicating the progress of each concrete production process instance. So, deviations can easily be tackled with since just the little flags had to be rearranged accordingly.

The company is profiting enormously from this unconventional way of process usage since they always can keep an overview onto all production processes and could track and supervise them.

This example shows that process "execution" should more appropriately be interpreted as process "usage". It also shows that not only the complete automation of process execution is useful. We call this kind of process usage external tracking.

External tracking and the checklist method are two unconventional ways of executing process models. Nevertheless, they fully fulfill their purpose to take advantage from process models.

At the end of this section it should be mentioned that there is a whole bunch of approaches that aim at the improvement of flexibility in process execution. [20] discusses and classifies some of them. Additionally, there are powerful approach to cope with dynamic development processes [14]. Also these approaches deal with some special kind of flexibility, among other things the flexibility of dynamically changing processes. We do not want to ignore all these approaches; they are very valuable and improve the flexibility of (conventional) process execution. However, in this paper we do not want to discuss approaches for flexible process execution alone but want to show that there are alternatives to (conventional) process execution approaches. These mostly stem from the introduction of domain specific, compact process modeling constructs.

3.4 Classification

Although we have introduced new interpretations of process execution and have shown how they could be enacted, it must be mentioned that the conventional process execution modes as they are supported by workflow management systems is still a valid candidate. There are processes that require a strict execution order which must not be violated. In such cases conventional workflow management systems provide a valuable platform for process execution.

Altogether, this section should show that other process usages than running them on workflow management systems is feasible and relevant. It will typically be the case that in one application domain a mixture of all these process execution modes will be relevant. Although we have investigated all four usage modes in detail, the remaining challenge is to combine them in one comprehensive application system. This could also be a motivation for researchers in general to think about alternative execution modes for processes.

In summary we want to depict the four introduced execution modes on a scale; this shows a broad spectrum of execution modes and should trigger researchers to look for more in order to more adequately meet the requirements of process applications. Fig. 9 shows the spectrum of the usage modes presented in this section. We span two dimensions: a first dimension distinguishing between strictness and flexibility with respect to process execution; a second dimension distinguishing between usages within a computer system (internal) and outside of a computer system (external). By "internal" we mean that most of the applications and the usage of the system take place on a computer system; "external" means that many/some actions occur completely outside a computer system (e.g. the wallpaper approach).

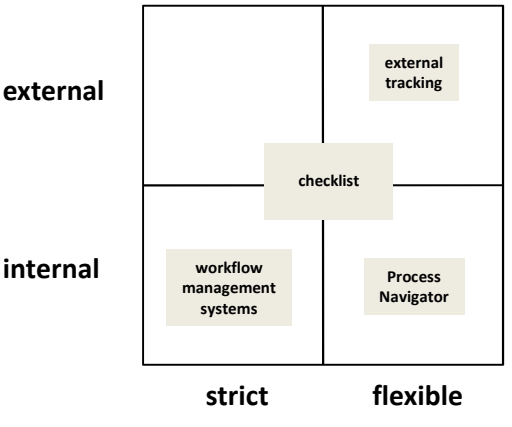


Fig. 9. Spectrum of process usage

Of course, workflow management systems are the handling process execution within a computer system and prescribe very strictly what to do. The ProcessNavigator relaxes the execution of processes but mainly aims at process executions within a computer system. The external tracking approach is most flexible and also leaves the

technical environment; however it is also possible to implement it on a computer system. The checklist approach can be configured both flexible and strict, and both intern and extern. So, it allows most degrees of freedom, depending on how it is implemented.

4 Conclusion and Outlook

At the end of this paper we want to summarize its main contribution. The purpose of this paper was not to introduce another piece of research. Rather, we wanted to stimulate the process management community to think about alternative and more powerful approaches to process modeling and execution.

Through presenting a couple of feasibility studies we already could show that there are valuable alternatives that are worth to be investigated further. We are convinced that these approaches are just the starting points for more appropriate ways of process management.

In detail, we want to motivate to contemplate the following research issues:

- Is there a set of elementary process modeling elements?
- How to develop domain specific models effectively and efficiently?
- How to find out which domain specific modeling primitives are optimal for a specific domain?
- What does "enactment" of a process model mean? Is it more than execution?
- How to make process execution more flexible?

These questions point to research issues which altogether aim at more adjustable and customizable process modeling language, since we regard adjustability and customizability as two of the key factors for the acceptance of process management.

References

[1] Program Evaluation and Review Technique, Wikipedia, http://en.wikipedia.org/wiki/Program_Evaluation_and_Review_Technique (retrieved: 2009-11-03)

[2] Critical Path Method, Wikipedia, http://en.wikipedia.org/wiki/Critical_path_method (cited: 2009-11-03)

[3] Kimball, R., Ross, M., Thornthwaite, W.: Kimball's Data Warehouse Toolkit Classics. John Wiley & Sons, Chichester (2009)

[4] Elmasri, R., Navathe, S.: Fundamentals of Database Systems. Pearson Education, London (2006)

[5] Object Management Group, Business Process Model and Notation (BPMN), Version 1.2, OMG Document Number: formal/2009-01-03, http://www.omg.org/spec/BPMN/1.2 (cited: 2009-11-03)

[6] Object Management Group, Business Process Model and Notation (BPMN), FTF Beta 1 for Version 2.0, OMG Document Number: dtc/2009-08-14, http://www.omg.org/spec/BPMN/2.0 (cited: 2009-11-03)

[7] Object Management Group, MOF 2.0 Specification, http://www.omg.org/spec/MOF/2.0/ (cited: 2009-11-01)
[8] Object Management Group: UML 2.2 Super- & Infrastructure. In: OMG (ed.) (2009), http://www.omg.org/spec/UML/2.2/
[9] Faerber, M., Meerkamm, S., Jablonski, S.: The ProcessNavigator - Flexible Process Execution for Product Development Projects. In: International Conference on Engineering Design (ICED), Stanford, CA, USA, August 24-27 (2009)
[10] Faerber, M., Jochaud, F., Jablonski, S., Stöber, C., Meerkamm, H.: Knowledge oriented Process Design for DfX. In: 10th International Design Conference, Dubrovnik, May 15-18. The Design Society (2008)
[11] Faerber, M., Jablonski, S., Schneider, T.: A Comprehensive Modeling Language for Clinical Processes. In: 2nd European Conference on eHealth (ECEH 2007) (2007)
[12] Fuentes-Fernández, L., Vallecillo-Moreno, A.: An Introduction to UML Profiles. UPGRADE - The European Journal for the Informatics Professional 5, 6–13 (2004)
[13] Hammer, M., Champy, J.: Reengineering the Corporation. In: A Manifesto for Business Revolution. Nicholas Breadley, London (1993)
[14] Heer, T., Heller, M., Westfechtel, B., Woerzberger, R.: Tool Support for Dynamic Development Processes (to be published in this book)
[15] Chen, P.: The Entity Relationship Model – Toward a Unified View of Data. TODS, 1, 1 (1976)
[16] Jablonski, S.: MOBILE: A Modular Workflow Model and Architecture. In: Proc. International Working Conference on Dynamic Modelling and Information Systems, Noordwijkerhout, NL (1994)
[17] Jablonski, S., Bussler, C.: Workflow Management: Modeling Concepts, Architecture and Implementation. International Thomson (1996)
[18] Jablonski, S., Volz, B., Dornstauder, S.: On the Implementation of Tools for Domain Specific Process Modelling. In: 4th International Conference on the Evaluation of Novel Approaches to Software Engineering (ENASE), Milan, Italy, May 9-10 (2009)
[19] Jablonski, S., Igler, M., Günther, C.: Supporting Collaborative Work through Flexible Process Execution. In: The 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2009) (2009)
[20] Jablonski, S., van der Aalst, W.M.P.: Dealing with workflow change: identification of issues and solutions. International Journal of Computer Systems Science & Engineering 15(5) (2000)
[21] Wegener, P.: Why interaction is more powerful than algorithms. Commun. ACM 40, 5 (1997)
[22] Lorenzen, P.: Dialogical Foundation of Logical Calculi. Constructive Philosophy (1987)
[23] Hughes/ Cresswell: An Introduction to Modal Logic, Methuen (1982)
[24] Tarski, A., Helmer, O.: Introduction to Logic. Dover Pubn Inc., New York (1995)

A Meta-Method for Defining Software Engineering Methods

Gregor Engels and Stefan Sauer

Universität Paderborn
s-lab – Software Quality Lab
Warburger Straße 100
D-33098 Paderborn, Germany
engels@upb.de, sauer@s-lab.upb.de

Abstract. Today's software systems demand for sophisticated software engineering processes and methods. Especially the globally distributed development of large software systems requires precise and documented methods, but also lightweight and agile methods need to have a precise foundation. Effort that is invested once in the methods can be systematically reused in projects. We describe MetaME, a meta-method for modeling and tailoring software engineering methods. It builds on a meta-model of software engineering concepts. MetaME combines ideas from meta-modeling and method engineering. The meta-method comprises a product dimension and a process dimension. When the meta-method is applied, software development concepts are paired with languages for their representation to form artifact types. In the process dimension of the software engineering method, software development tasks are described as operations that act upon the artifacts. These tasks are performed as activities in the method's process workflow model. Tools can then be built that use the artifact model as the foundation of their repository structure and the task and workflow models as the basis for the supported functionality.

Keywords: Method Engineering, Software Engineering Method, Meta-Model.

1 Introduction

Software systems such as business information systems are constantly growing in size and complexity. At the same time, they need to be produced in dependable quality while their development shall be cost and resource effective. To meet all these requirements, the development of software systems demands for sophisticated software engineering methods and processes.

In this work we use the term *software engineering method* to denote the full set of elements needed to describe a software development endeavor, such as a software development project, in all relevant aspects. This does not only cover the software development process and its contained activities, but also the artifacts that are to be produced, the tasks that need to be performed to achieve the development goals, the roles in an organization that participate in the development, the tools, techniques and utilities that are employed, as well as relationships between these concepts.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 411–440, 2010.

To obtain such software engineering methods, their own development should be done systematically and have a sound methodical foundation. This is the objective of *method engineering* [Bri96]. Method engineering is an engineering discipline that deals with the development of methods, techniques, and tools for the development of software systems. It started in the area of information systems in the 1990s (e.g. [Gut94, Bri96]), but was taken up by software engineering in the following (see e.g. [NFK94, Rol09, HR10]). Method engineering aims at providing a framework for defining and tailoring system and software engineering (SE) methods. It allows us to model and analyze even complex software engineering methods in a systematic way.

Different software engineering methods and processes have been proposed and are in use for different purposes, such as the Rational Unified Process (RUP) [IBM07] or agile methods like SCRUM [SB02] and many more. However, it is widely recognized that such standards are often too generic to be directly applicable and thus must be tailored to the problem at hand (see e.g. [Wie03]) before they can effectively be employed. It becomes also necessary to develop new methods due to the advent of new development paradigms; or domain-specific methods that account for the specifics of a certain domain like business information systems or business intelligence systems; or for a particular delivery model such as global software development [SSEB10]. Hence there is still a need to derive, evolve and develop new software engineering methods.

Tailoring of methods is necessary since there exists no standard method that perfectly suites all types of projects in all domains. It is also not reasonable to develop a new method every time when a context-specific method is needed. It is much more economic to tailor existing methods to the current development context and situation.

A number of method engineering approaches have been proposed that especially deal with the development of methods for a particular situation, which is known as *situational method engineering* (see e.g. [RBH07, BKPJ07, HR10]). Mechanisms for reuse and adaptation play an important role in this field. In addition, component-like concepts that support modularity of methods are often used, such as viewpoint templates [NFK96], method fragments [Bri96], or method chunks [Rol09]. In recent publications, even the use of method services and the notion of method-as-a-service are discussed [Rol09].

Another line of research has been focusing on the use of meta-modeling for the representation of method contents. A number of meta-models have been developed in result, see e.g. [HG05, GMH05, BG08, JJM09]. With the Object Management Group's (OMG) Software & Systems Process Engineering Meta-Model (SPEM) [OMG08] and the ISO/IEC 24744 Software Engineering – Metamodel for Development Methodologies [ISO07] even two standards for describing the content of software engineering methods arrived.

The advantages of a meta-model-based approach are manifold: First, meta-modeling provides a formal and precise foundation for the specification of software engineering methods. Secondly, software engineering methods can be compared on the formal basis provided by the meta-model, acting as a reference framework. Third, the formalization provides a precise foundation for the development of tools and utilities that support the use of the method. Fourth, the meta-model can be employed

for analyzing a software engineering method for certain properties such as consistency, conformance, etc. Last but not least, the meta-model provides a formal basis for tailoring. Changes to software engineering methods can be traced back to the meta-model and checked for their conformance and consequences, since methods are instances of the meta-model.

In our analysis we have observed a number of reasons why the current approaches still have some shortcomings. The first and most obvious point is the lack of a process definition that specifies how to develop a software engineering method based on the meta-modeling approach; in other words, how a method engineer should instantiate the meta-model. Neither do they define the tasks that are needed for method engineering nor a process workflow to follow. Strictly speaking, meta-models like SPEM do not define a meta-model for software engineering methods, but a meta-model for method descriptions. That means, they define a modeling language for software engineering methods.

Secondly, most approaches lack a sound integration of the product and the process aspect of a software engineering method in a coherent, yet manageable meta-modeling architecture. For example, the OMG favors to complement SPEM [OMG08] with the UML meta-model for the definition of the models to be produced, and some behavior modeling formalism like BPMN to define the behavior of the methods process part. ISO 24744 proposes to combine the different aspects by the use of powertype patterns and clabjects [ISO07, GH08]. ISO thus provides a sophisticated, yet challenging formal approach to address the integration issues that contrasts the ideas of strict meta-modeling as discussed in [AK01].

Additionally, although most approaches offer some means to interrelate the process and product aspects at least on a high level of abstractions – such as products that are used or created by tasks and activities in the process model, or roles that are responsible for products or perform tasks – they lack the possibility to model complex patterns of interlinked structural and behavioral models. For example, SPEM allows the method engineer to define work products (artifacts) and work elements such as tasks, processes, activities, and steps of tasks. Furthermore, state models can be assigned to work product definitions; and the states and transitions of the work products' state models can be related to work elements. But this linking is restricted to the work product lifecycle of individual work products. What cannot be expressed in a formal way is the effect of work elements on the artifact object structure, i.e. the network of interrelated work product instances. Instead of coding this by states, we wish to have an explicit mechanism for defining such transformations.

Based on this observation, it is the objective of our work to combine method engineering, meta-modeling and ideas from language engineering for the development of software engineering methods and to present a concise *meta-method* in this paper for defining and tailoring software engineering methods. The meta-method is designed to support the definition of software engineering methods as well as the tailoring of software engineering methods for particular domains and projects, i.e., as an approach for situational method engineering.

The meta-method must cover both the product and the process dimension of the software engineering method. In the product dimension, the method engineer must specify which artifacts are to be produced in the course of applying the method and how these artifacts are related. In the process dimension, it must be defined how to

proceed for producing the artifacts and what needs to be accomplished – and by whom – in getting from one artifact to the other. The former can be covered by process or workflow models. The latter can be achieved by the use of transformations. Such transformations can be executed as manual development tasks or by automated tasks as part of the software development process. Thus, we define the product part of the software engineering method in an artifact model and the process part of the method by workflow models and task models. The effect of task execution will be modeled by the use of transformations that operate on the artifact model.

We propose transformation rules that operate on instances of the product model. These rules can also make reference to the state model of individual work product elements, but show also their attribute values and links in the pre- and post-conditions of the transformation rule.

We use the same set of general specification means also to define the meta-method, being itself a method for the development of methods. We use a product meta-model from which software engineering methods are instantiated. The process dimension of method engineering is described by the workflow model and the task model of the meta-method. The workflow model defines how to proceed in order to define a software engineering method. The task model defines the tasks that need to be accomplished, again in the form of transformations.

The remainder of this paper is structured as follows: In Sect. 2 we give a very brief introduction to the core concepts of software engineering methods. Meta-modeling is introduced in Sect. 3 as a methodological means for developing software engineering methods. In Sect. 4, we deal with the domain of method engineering and present OMG's SPEM [OMG08] in some detail. Section 5 constitutes the core of this paper where we present our meta-method for method engineering of software engineering methods. We present the product meta-model as a general information model of method engineering as well as a process for developing software engineering methods. Steps 3 and 4 of that process that are concerned with the development of an artifact meta-model and the software process model are looked at in detail. There we also introduce the idea of specifying software engineering tasks as transformation rules on the product model. The issue of tailoring in the framework of our meta-method for the engineering of software engineering methods is described in Sect. 6. Section 7 concludes this paper with a summary and outlook.

2 Core Concepts of Software Engineering Methods

In order to manifest a common understanding of the fundamental concepts and central technical terms that are used throughout this paper, we will introduce these terms in this section. We look in the domain of software engineering, being the domain of the software engineering methods, and the domain of method engineering, being the domain of the meta-method.

From the software engineering perspective, the central concept of our approach is the software engineering method. We define *software engineering method* as a systematic procedure or technique of doing work in software engineering in order to reach a certain goal and/or produce a defined set of software artifacts. Software engineering

methods structure, coordinate and document the development processes and activities as well as the produced artifacts (also called work products or work items). Software engineering methods are often also called software process models by other authors. However, we think the term process may lead to misunderstandings, since a software engineering method contains more than just a process definition. Rather, the software engineering method is concerned with the processes *and* products of software engineering. Such products are e.g. different kinds of software models that themselves may have a complex structure that needs to be specified in a comprehensive software engineering method.

Software development processes are defined by IEEE Standard 610.12 as follows [IEEE90]: "The process by which user needs are translated into a software product." The process involves *activities* such as translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use. Activities may overlap or be performed iteratively. From this definition we can derive the understanding, that processes are made from activities that are executed in some order. In accordance with [OMG08], we define that the processes of a software engineering method are hierarchically composed from activities. More specific concepts may be defined to capture specific kinds of sub-hierarchies on intermediate layers such as iterations and phases of development processes, like they are defined in RUP [IBM07]. These compositional structures can be seen as specific kinds of (composite) activities. Processes and activities thus define the the *process structure* and *workflow* of a software engineering method.

Two other terms that are closely related to the process are software development cycle and software life cycle. They refer to the *temporal* aspect of the software engineering process. According to IEEE Standard 610.12 [IEEE90], a *software development cycle* is "the period of time that begins with the decision to develop a software product and ends when the software is delivered." The development cycle typically includes *phases* such as a requirements phase, design phase, implementation phase, test phase, and sometimes, installation and checkout phase. The phases of the cycle – alike the activities of a process – may overlap or be performed iteratively, depending on the software development approach that is employed. In contrast, the *software life cycle* is defined as "the period of time that begins when a software product is conceived and ends when the software is no longer available for use." The software life cycle thus goes beyond the software development cycle. It extends the development cycle with additional phases: The software life cycle typically also includes a concept phase upfront, and subsequent to development an installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. Again, these phases may overlap or be performed iteratively.

A complete set of software engineering methods ideally covers the full software life cycle, at least it should cover the full software development cycle. If it only covers the software development cycle, the term software development method is as well appropriate.

A software engineering method comprises a set of concepts that are required for its definition. It is commonly agreed that a software engineering method has to cover at least three main aspects by its provided concepts (see e.g. [GH08]): the work products that are created and used, the process to follow, and the producers that are involved.

Although different authors define different sets of concepts, some of them are commonly used (although sometimes using different terms with varying semantics) and can be seen as the agreed minimal set (sometimes further differentiated): roles (and/or people), processes and activities (and/or tasks), work products (or artifacts), tools (software tools and other utilities). The authors of the Unified Software Development Process [JBR99], for example, identify five main entity types for software development methods: process, people, project, product, and tools (as depicted in Figure 1), thus adding the concept of project explicitly.

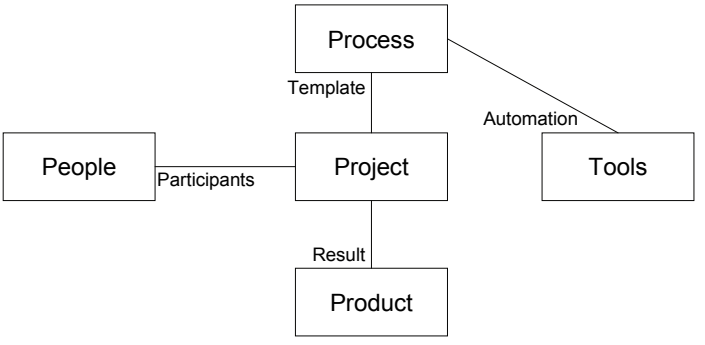


Fig. 1. The core concepts that make up the Unified Software Development Process (redrawn from [JBR99])

Gutzwiller [Gut94] has developed a meta-model for development methods and process models. It requires five general elements for describing a method: activities, roles, work products, techniques, and an information model (termed 'meta-model' in the original work), see Figure 2.

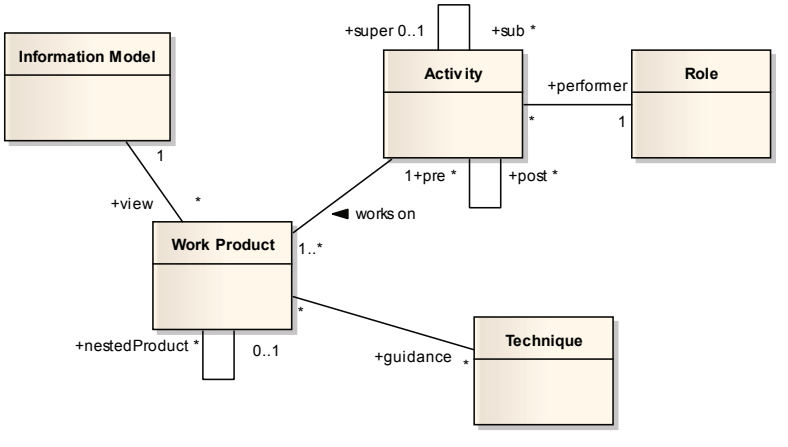


Fig. 2. The core concepts of a software engineering method according to [Gut94]

According to [Bal98], a software engineering method defines the following aspects:

- workflow of development process,
- activities that are to be executed,
- definition of work products (or product parts) with respect to content and structure/layout,
- completion criteria for work products (or product parts),
- required skills for performing tasks,
- responsibilities and capabilities of workers,
- standards, guidelines, techniques and tools that are to be employed.

The Rational Unified Process [IBM07] distinguishes between the static and the dynamic aspect of a software development process. The static aspect describes the process structure that is built from activities, workflows, artifacts, and workers. The dynamic aspect of the process as it is enacted covers the temporal domain by the concepts lifecycle, phase, iteration, and milestone. Workflows group activities logically. Activities comprise activity steps. Artifacts are models, model elements, documents, source code, and executable software. Deliverables are a kind of artifact. Workers are a role concept for people and resources, having a responsibility relation with artifacts.

SPEM [OMG08] distinguishes between the method content and the process of a systems and software engineering method. The core concepts defined in SPEM [OMG08] for the method content are: work product, task, role, tool. The core concepts of the process are: activity, milestone, work product use, task use, and role use. Processes are defined in a general hierarchical breakdown structure with inner nodes being activities. The workflow is defined with temporal relationships, called work sequences. Work product use, task use, and role use are used to make reference to the corresponding method content elements.

ISO 24744 [ISO07] distinguishes between ten key concepts. Five of them are assigned to the method domain: language, notation, constraint, guideline, and outcome. The other five are assigned to the endeavor domain: stage, work unit, work product, model unit, and producer. Action is used to relate tasks to work products.

Stage is further specialized in instantaneous stage (e.g. milestone) and stage with duration (e.g. time cycle, phase, build). Specializations of work unit are task, technique, and process. Subtypes of work product are composite work product, software item, hardware item, document, and model. The elements of models are captured by model unit. Role, person, team as well as tool are subtypes of producer.

Guidelines can be associated with any methodology element. The observable results of performing any kind of work unit are given by the class `Outcome`. Constraints are aggregated by action kinds and are specialized in preconditions and post-conditions. Notation is associated with both document kind and language. The relation with document kind denotes that a document kind uses one or more notations. The association between notation and language states that multiple notations can depict a language and, vice versa, a notation can depict multiple languages. Language aggregates model unit kind to denote that any model unit kind is always defined in the context of at least one language, while a language can be the context for one or more

model unit kinds. A direct association between model kind and language allows a method engineer to express which language is used for one or model kinds.

From this brief discussion of concepts provided in different meta-models or employed in software engineering methods one can already see the ontological diversity of the approaches. In an attempt to bring them closer together we have contrasted a selected set of core concepts of the different approaches in Table 1. In the last column we have added the matching concepts of our approach that is named MetaME – meta-method for method engineering. They will be further discussed in Sect. 5.2.

Table 1. Comparison of core concepts for software engineering methods

SPEM	ISO24744	RUP	MetaME
	Stage	Discipline	Domain Discipline
			Concept
Work Product Definition, Work Product Use	Work Product (Model, SoftwareItem, HardwareItem, Document) ModelUnit	Artifact (Model, ModelElement, Document, SourceCode, Executable)	Artifact
Work Definition	WorkUnit	Work	Work
Activity	Process	Workflow Activity	Process Activity
Task Definition, Task Use	Task	Task	Task
Step	Action	ActivityStep	ActionStep
			Transformation
(Phase, Iteration) as Kind	TimeCycle, Phase, Build	Lifecycle, Phase, Iteration	Phase
Milestone	Milestone	Milestone	Milestone
Role Definition, Role Use	Role	Worker	Role
Tool Definition	Tool		Tool Utility
	Language, Notation		Notation
Guidance	Guildeline		Guidance
	Technique		Technique
	Constraint		Constraint

Comparable meta-models can also be derived from existing software engineering methods, and of course, there exist many more of such meta-models for software engineering methods, like the one mentioned in Sect. 1. From the literature and our own project experience, we have derived the meta-model for software engineering methods that will be presented in Sect. 5.2. Some fundamentals of modeling and meta-modeling will be discussed in the next section.

3 Meta-Modeling for Software Engineering Methods

A promising approach towards the systematic and structured development of software engineering methods is the use of meta-modeling techniques for specifying the software engineering methods.

A *model* is, according to scientific theory, a representation of a natural or artificial original that focuses on those characteristics and properties of the original that are relevant for the given purpose of modeling, and abstracts from irrelevant properties. The purpose depends on both the creator and user of the model, and the intended use of the model. In an engineering process, models are used for specification, documentation, and communication. They are themselves objects of processing and transformation, and are a foundation for decision making, analysis, validation, verification, and testing. Models can be built upfront or retrospective in terms of forward engineering or reverse engineering, respectively.

A *meta-model* is a model of a model. *Meta-modeling* is, according to [GH08], "the act and science of creating meta-models, which are a qualified variant of models." The specialty of a meta-model is that the information it represents is itself a model. The meta-model's concern is the modeling itself. In the domain of method engineering, the meta-model is a model of a software engineering method.

In object-oriented meta-modeling, a model conforms to its meta-model in the way that it is an instance of the meta-model. A software engineering method can then be understood as a model that is an instance of this meta-model. The meta-model together with the definition of the semantics of software engineering concepts that are contained in the meta-model define an ontology for software engineering methods.

Meta-models are commonly used for defining modeling languages. However, they may also be used for defining in a wider sense the process of modeling (compare [Str98]). Meta-modeling is already widely used for defining software modeling languages as well as models of for software development methods, e.g. in the case of UML [OMG09a, OMG09b] and SPEM [OMG08] or ISO 24744 [ISO07], respectively. UML is a standard language for modeling software systems. SPEM and ISO 24744 are languages for describing software development methods and process models, i.e., meta-models for method descriptions.

For the *domain of method engineering*, we adopt the definition from [GH08]: A meta-model is a domain-specific language that is intended to represent software development methods. ISO 24744 defines the meta-model in the context of method engineering to be the "specification of the concepts, relationships and rules that are used to define a methodology" [ISO07]. (Note: methodology is synonymously used to method here, denoting the "specification of the process to follow together with the work products to be used and generated, plus the consideration of the people and tools involved" [ISO07].)

The OMG has defined a four-layer meta-model reference architecture in the Meta Object Facility (MOF) [OMG06] that builds on the concepts of object-orientation and is commonly used in meta-modeling (see Fig. 3). According to this meta-model hierarchy, we can characterize the levels for the domain of method engineering as follows:

M0 (Runtime layer) – M0 denotes the lowest level of the MOF 4-layer meta-model hierarchy. In this layer, objects of the real world are denoted that exist at

execution time of the modeled system. More generally, M0 represents the area of concern, which may be business, software engineering, or method engineering. In the domain of method engineering these are the concrete objects that are produced or modified during the lifecycle of a concrete software engineering endeavor.

M1 (Model layer) – M1 is the layer where user models are located. Reality is modeled in a modeling language, such that elements of M0 are instances of elements in M1. In the domain of method engineering, the model of the method is allocated on this level.

M2 (Meta-model layer) – M2 is the layer where meta-modeling takes place. It contains meta-models (models of models) such as the UML meta-model or SPEM which define modeling languages for describing the user models of layer M1. Elements of user models from M1 are then instances of meta-model elements of layer M2. This level holds the meta-method's model in the domain of method engineering.

M3 (Meta-meta-model layer) – M3 is the highest level of the 4-layer meta-model hierarchy. Meta-meta-models are defined at this layer. They are used to describe the meta-models on layer M2. In the MOF hierarchy, the Meta Object Facility itself is defined on this level. Defining method engineering within an object-oriented meta-model hierarchy, we use MOF for the domain of method engineering on this level as well.

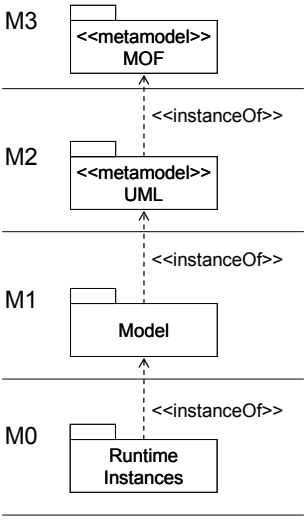


Fig. 3. General 4-layer MOF meta-model hierarchy

Based on this meta-model hierarchy and the characterization of software engineering methods, we define the meta-model of our meta-method (M2). It contains the meta-classes for the important concepts that are required to model a software engineering method. We will build on this four-layer meta-modeling architecture as the guiding principle in the definition of our meta-method's architecture. This will be explained in Sect. 5.1.

4 Method Engineering

After having introduced the basic ideas of meta-modeling in the previous section, we will now continue with the topic method engineering, and then show how meta-modeling has been applied for method engineering in this section. SPEM will be briefly explained as an example meta-model.

4.1 Introduction to Method Engineering

Method engineering has been an active research area in the field of information systems engineering since the early 1990s. Method engineering is concerned with the systematic construction of software development methods [Gut94]. [Hey95] defines method engineering as the systematic and structured process of development, modification and adaptation of methods by describing the components of the method and their relationships. In general, it is the objective of method engineering to formalize the use of methods for systems development [HR10]. More precisely, method engineering can be defined as the *engineering discipline to design, construct and adapt methods, techniques and tools for the development of (information) systems* [Bri96, HR10]. The objective of method engineering is to develop a methodical approach for systems development in a given context (and situation) such as an organization or project.

Method engineering mainly addresses two perspectives: a) the systematic development of methods and b) the enactment and execution of methods. Both aspects may themselves be supported by dedicated tools, such as a method development environment or a workflow engine.

Applying method engineering to the domain of software engineering methods provides a number of advantages:

- method engineering provides a methodological framework and conceptual infrastructure for method knowledge,
- method engineering supports a systematic development of SE methods,
- by providing specific means for method adaptation, methods can be adapted to a particular situation and context of use *(*cf. *situational method engineering*, see [HR10] for a recent survey),
- concepts of method modularization, reuse and configuration [BKPJ07] can be used to assemble methods from methodical building blocks, such as *viewpoint templates* [NFK96], *method fragments* [Bri96], *method chunks* or *method services* [Rol09],
- the meta-models that are used for the definition of methods enable analysis and comparison of methods, even quantitatively, by the use of an accompanying quality model and metrics,
- method engineering can ease reuse and provide means for compositional method development, and method integration,
- method engineering builds a sound basis for tool support, e.g. computer-aided software engineering (CASE) tools that may be built by using Meta-CASE tools.

The product of a method engineering process is a method. In the context of this work, the users of this product are software engineers who develop software-based systems.

The lifecycle of a method is similar to the lifecycle of a software system. We can interpret a method as a conceptual system for system development. Method engineering manages and controls this method lifecycle and may even itself be computer-supported by its own software system, a computer-aided method engineering (CAME) tool [Bri96]. The general overall lifecycle model of method engineering is depicted in Fig. 4. Once the domain of discourse has been identified (being software engineering in our case), the requirements for the method are analyzed. It follows a multi-stage development process. Then the method is deployed, used, and evaluated in order to start another evolution cycle.

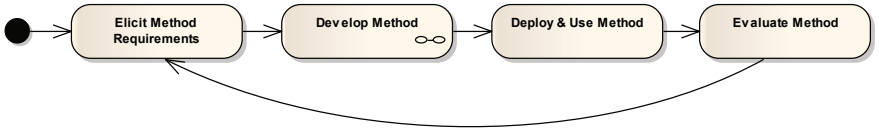


Fig. 4. Lifecycle of a software engineering method

4.2 Meta-Modeling for Method Engineering

Meta-modeling has been identified as a promising means for method engineering. Several meta-models have been defined in the literature by different authors, see e.g. [JJM09], [BG08], [GMH05], [HG05]. Two standards also exist that use meta-models for the definition of software development methods: ISO 24744:2007 Software Engineering − Metamodel for Development Methodologies [ISO07] and SPEM, OMG's Software & Systems Process Engineering Meta-Model Specification [OMG08]. The latter provides a meta-model as well as a UML profile for the specification of software development methods. We present SPEM as the most acknowledged meta-model as an example in the following.

The engineering of software engineering methods happens within three domains: meta-method engineering, method engineering, and software engineering. Each of them corresponds to a distinct level of abstraction. These levels of abstraction correspond to the layers of the meta-modeling hierarchy depicted in Fig. 3. Different tasks of SE method engineering have to be performed in the three domains for producing the required products on the different levels of the meta-model hierarchy. These tasks are performed by dedicated roles according to our meta-method (see Figure 5). The meta-method engineer is responsible for defining the meta-method for method engineering (M2) in the meta-method engineering domain. This meta-method is applied by the method engineer in the method engineering domain in order to develop a concrete software engineering method (M1). The software engineering method is then used by software engineers in the software engineering domain for developing the software system in a real software development project (M0).

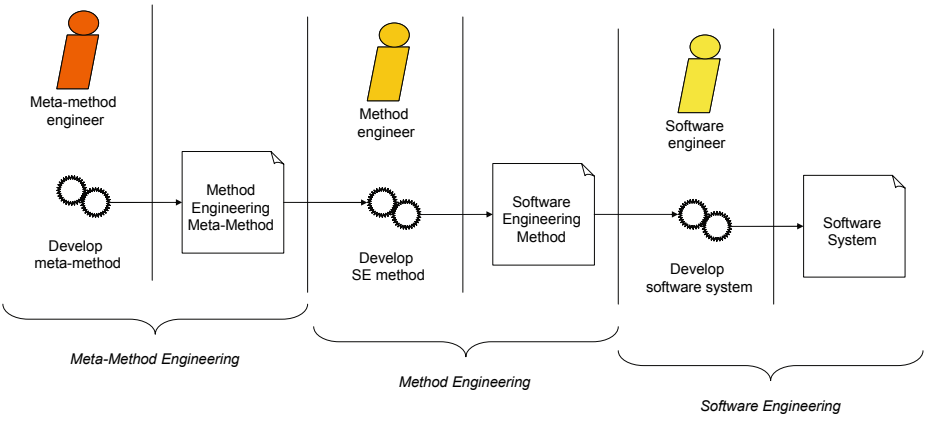


Fig. 5. Dedicated roles are responsible for producing the work products on the different layers of the meta-model hierarchy

4.3 SPEM

The Software & Systems Process Engineering Meta-Model (SPEM) [OMG08] is intended for defining software and system development processes. The OMG characterizes SPEM as "a process engineering meta-model as well as conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes" [OMG08].
SPEM is a meta-model that is based on MOF and UML. Objective of SPEM is to provide description elements for software development methods that are independent of parameters such as the development paradigm being deployed (e.g. agile, architecture-centric or code-centric, test-driven or model-driven software development), the degree of formalization or the cultural background. The set of language elements is intended to be minimal for this purpose.

SPEM thus really is a meta-model for describing software engineering methods, and not a method engineering method, since it does not contain a method engineering process definition. Furthermore, SPEM has not been intended to be a process modeling language for software development processes nor does it even provide its own behavior modeling mechanisms. It provides neither a concrete process modeling language nor guidance for selecting such a process model. It only provides the interface for docking a complementary behavior modeling mechanisms. Hence, SPEM is just a description language for software engineering methods.

An important concept of SPEM is the separation of the method content and the development process. Method content denotes descriptions of how to achieve particular development goals. Such contents are independent of their use in a specific development process. The method contents are then applied within a process and brought into a temporal order.

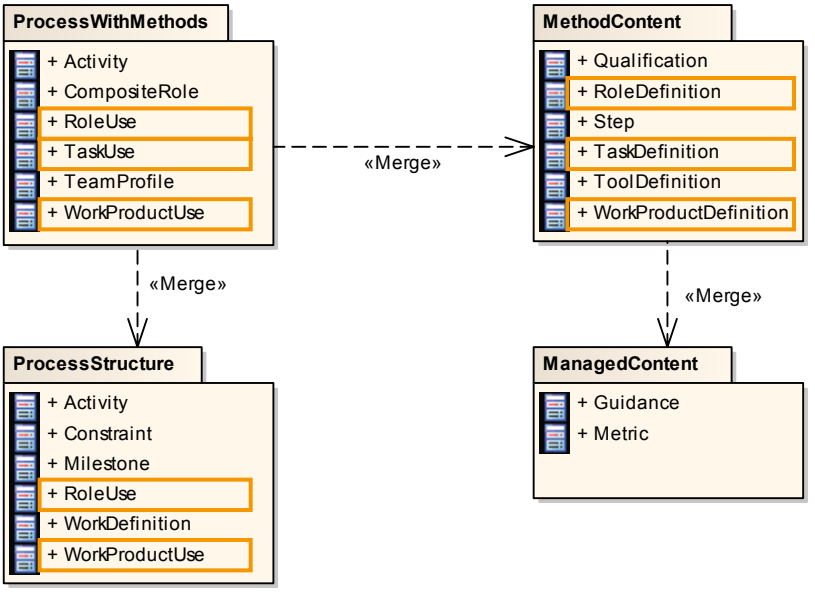


Fig. 7. Separation of method content and development process in SPEM

Figure 7 visualizes the separation of method content and process. It shows four meta-model packages of SPEM including the meta-classes for the most relevant concepts. It can be seen that elements such as work products, tasks, and roles are defined as method contents (highlighted on the right-hand side) and then applied in the process part by work product use, task use, and role use (highlighted in the two packages on the left-hand side). What can be seen from this meta-model excerpt is that the package `ProcessWithMethods` actually integrates process structure and method content by its respective merge dependencies. Furthermore, guidance and metric from the package `ManagedContent` are available as general concepts in the merging packages too.

After having presented the foundations from meta-modeling, method engineering and the use of meta-modeling in method engineering – using SPEM as an example – we will now present our meta-method for the definition of software engineering methods in the next section.

5 MetaME – A Meta-Method for Software Engineering Methods

Based on the foundations of method engineering and meta-modeling, we have developed MetaME, a meta-method for the engineering of software engineering methods. We deploy a four-layer meta-model architecture in order to define the meta-method. The meta-method defines the language to cover the *product* of the method

engineering process, i.e., the method description, and the *process* that is used to build a software engineering method. We first present the meta-model architecture that we use for integrating the product models and the process models across the different meta-modeling layers in Sect. 5.1. We then describe the respective meta-models on the meta-modeling layer M2 and their integration (Sects. 5.2 to 5.4). Sections 5.5 and 5.6 exemplify the artifact and the process model on the method level M1 that instantiate the meta-model from M2. Finally, the use of transformation rules in the work model is shown in Sect. 5.7.

5.1 Meta-Model Architecture of the Meta-Method

We build on meta-modeling in the definition of our meta-method for method engineering. However, we have discovered that simply employing object-oriented meta-modeling in the sense of the Meta Object Facility has some shortcomings (see also Sect. 1). MOF is restricted to defining the structure (abstract syntax) of a modeling language. It does not comprise any means for modeling behavior. Furthermore, it only allows an object-oriented type-instance relationship between classes and objects of directly adjacent meta-modeling layers.

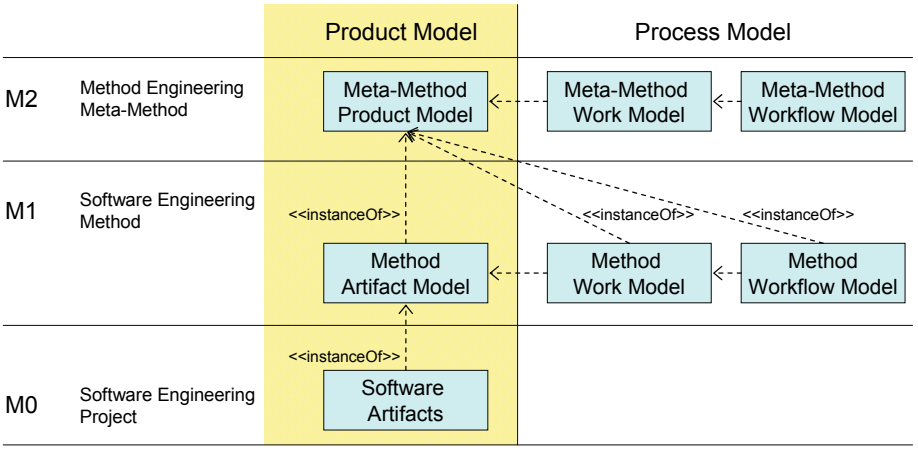


Fig. 8. A method consists of a product model and a process model; this applies to the meta-method on level M2 as well as to the software engineering methods on level M1

To define a method, we combine the method's product model (shaded in Figs. 8 and 9) with its process model. The process model is composed of a work model that defines work elements such as activities and tasks and a workflow model that defines the temporal ordering of activities. We apply this method pattern on the meta-method level and the method level as shown in Fig. 8.

However, while the meta-method process model (M2) must be an instance of a process meta-model (M3) to have execution semantics (see Fig. 9), all parts of the method (M1) are defined as instances of the meta-method product model (M2), since

the complete method is the product of the method engineering process. Yet, the method process model (M1) must also be an instance of the process meta-model (M3), since it is a process model itself. Yet, this instantiation relationship skips the M2 level and thus is not compliant with strict meta-modeling as described in [AK01].

We solve this problem by bootstrapping the process meta-model into the meta-method product model with a <<merge>> relationship (see [OMG06]), like this was done for MOF and UML too. Thereby, we also convert between ontological and linguistic meta-modeling, since the meta-method product model on M2 as well defines the method modeling language. The method is engineered and modeled by instantiating the meta-method product model (M2) and enacting the meta-method's process model. This relation is represented by the dependency of type <<producedBy>> between the method and the instance of the meta-method process model (in level M1). The same pattern applies in the M0 level for the production of the development project's software artifacts.

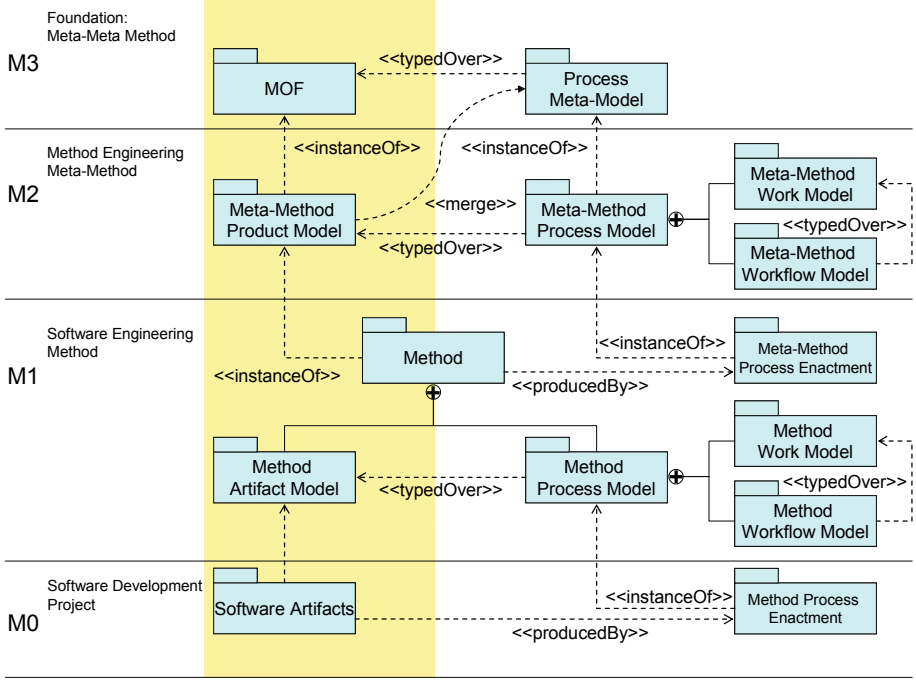


Fig. 9. Applying the meta-modeling approach for the engineering of methods

Figure 10 shows a focused view of the product model dimension across the layered architecture. It can be seen, that we distinguish between the method engineering domain (left-hand side) and the software engineering domain (right-hand side). The former is concerned with the engineering of software engineering methods, the latter with the engineering of software systems. Nevertheless, these two hierarchies are interrelated. First, both hierarchies are founded in the Meta Object Facility in their respective M3 layers. Secondly, the software engineering methods that are developed

in the method engineering discipline are applied in the software engineering discipline. For example, the product model of the software engineering method (M1 of method engineering domain) contains an artifact model in which the artifact types of the method such as `Task` and `Class` are defined. Instances of these artifact types are created as the concrete objects in a software development project. Additionally, these artifact types are transferred to the software engineering domain where they are deployed in the software system meta-model (M2 of software engineering domain) as the elements of a modeling language for the software system model (M1 of software engineering domain). Figure 10 shows an example. Such a software system model is produced in a concrete software development project (M0 of method engineering domain) which in turn instantiates the software engineering method. As a consequence, the two domains coincide (with a switch of layers) and must be either integrated or coupled.

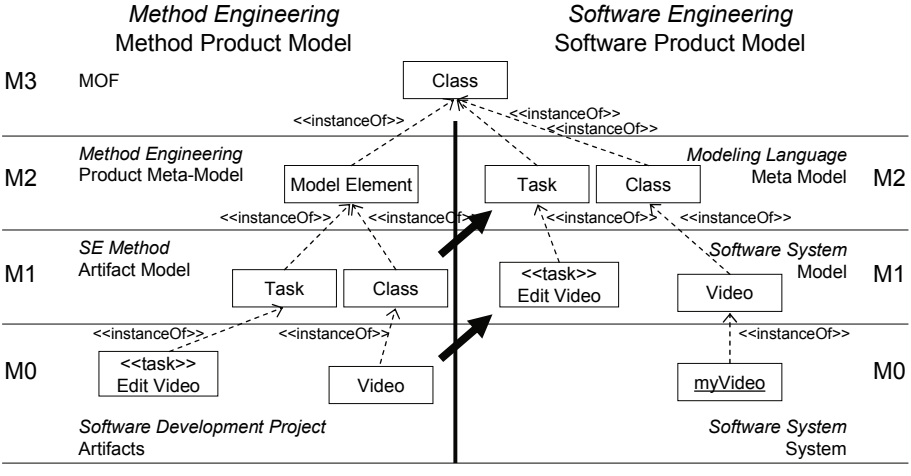


Fig. 10. Example for the relationship between the artifact model of a software engineering method (M1, left) and the meta-model of a modeling language (e.g. UML) used in software development (M2, right)

Based on this integrated meta-modeling architecture of product and process models, we describe in the following which constituents make up the meta-method in the different parts of the meta-model hierarchy. We start on the meta-method level M2 (Sects. 5.2 to 5.4) and will then exemplify its instantiation on the method level M1 in the remaining sub-sections of Sect. 5.

5.2 Method Engineering Meta-Method: Product Model

The meta-method for method-engineering on layer M2 of the SE method engineering domain is a method itself. Therefore, all relevant aspects for defining a method need to be defined for the meta-method as well: product model and process model. The product model (see Fig. 11) of the meta-method defines the fundamental

types of elements within software engineering methods. They can basically be bootstrapped to the method engineering meta-method as well. In addition, we will briefly explain the process of how to develop a software engineering method according to the product model in the next section.

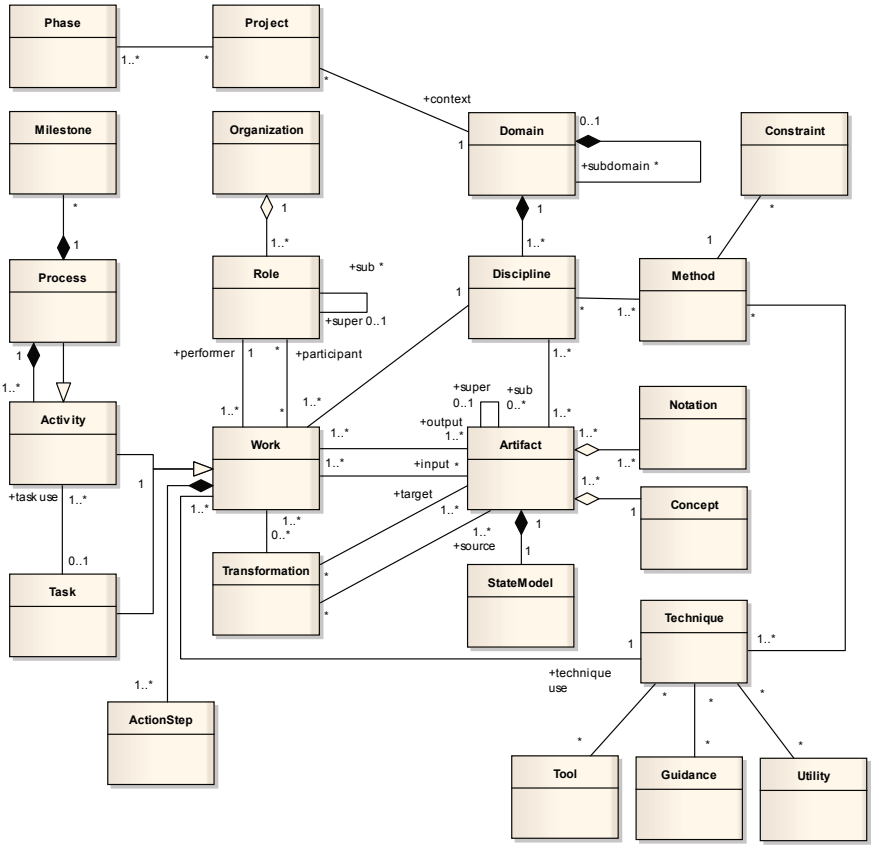


Fig. 11. Core domain concepts of software engineering methods defined in the meta-model of our meta-method

Figure 11 depicts the core concepts that are required for describing a software engineering method (Note: it is not the complete model). We have graphically structured the meta-class diagram for the core aspects that have to be covered (see Sect. 2). In the upper right, one can find the method content defining the structure of the method (domain, discipline, artifact, concept (semantics), notation (syntax)), and the general concept of constraint. On the left-hand side, the process dimension is covered by the concepts process, activity, task, action step, role, transformation, phase, milestone, etc.). At the bottom end, techniques, tools, and practices of the methods that provide guidance on how to accomplish the tasks and produce the work products are depicted (technique, tool, utility, guidance, etc.).

Domain captures the engineering or application domain for which a method is to be specified. Domains can be further decomposed into a set of disciplines. Artifact types are assigned to disciplines in which they are created or used. Each artifact type combines a concept and a set of notation elements. The lifecycle of an artifact can be modeled with a state model. Techniques are associated with methods to which they belong and/or to activities and tasks in which they are used. Tools, utilities, and guidance are assigned to techniques.

Work elements are specialized into tasks and activities. They can be further decomposed into atomic action steps. Work is also related with role to indicate its responsible performer as well as participants. Roles can be part of organizations. Transformations are associated with work elements as well, but also relate to the artifacts they use (source) or produce (target).

Processes are hierarchically composed from activities and can also include milestones (not shown is the relation between milestone and the corresponding artifacts that need to be finished to reach the milestone). Finally, the temporal domain of the endeavor is modeled by the project being associated with phases.

5.3 Method Engineering Meta-Method: Process Model

In Fig. 12, the fundamental process of the meta-method for developing software engineering methods is defined. This process model belongs to the meta-method process model (M2) in Fig. 9. It refines the composite activity "Develop Method" in the method lifecycle process depicted in Fig. 4. The numbers in Fig. 12 correspond to the steps (activities) of the meta-method process in order to produce a software engineering method by instantiating the meta-method's product model. The process model is thus typed over the product model. Steps 1 to 5 correspond to the steps that we have described in [ESS08]. The first step is of foundational character:

0. **Define domain and disciplines**: The domain is software engineering methods in our case, and disciplines are used to further structure the software engineering method into areas of concern, such as the disciplines of the Unified Software Development Process [JBR99] (compare Fig. 12).

In [ESS08] we have proposed a meta-method for defining software development methods based on domain models of software engineering concepts and artifact types. While the development of the domain models (steps 1 to 3, see [ESS08] for details) is the focus of that article, the definition of the process and the assignment of tools on top of these models are only sketched. The meta-method in [ESS08] has been applied, analyzed and extended in [Sta09]. We have revised those five development steps in the course of this work:

1. **Produce domain model of software engineering concepts**: In the sense of a product model, the domain model of software engineering concepts is set up and organized according to the identified disciplines (in the form of packages that may be hierarchically nested). Such disciplines may as well correspond to levels of abstraction (requirements, analysis, design, etc.) or views (partial models) of the

system (requirements model, analysis model, design model, etc.) Core activities are the definition of SE concepts and assigning them to disciplines.

The concepts can be further classified according to the aspect they model, for example, structure, behavior, or context. Concepts can be engineering or management related.

Relationships between concepts are added such as composition and aggregation relationships, dependencies, and associations. Refinement is an important type of relationship for describing forward engineering methods: a set of concept instances on a lower level of abstraction (partially) refines a set of concept instances on a higher level of abstraction. For example, a business use case can be refined by a system use case, and the system use case can be refined by a set of activities. Another example is a conceptual component in a system specification that is refined into a set of logical components within a software design.

The meta-model representation is accompanied by a glossary that contains an entry for each meta-model class. Each entry holds a description of the semantics, purpose and properties of the concept and relationships to other concepts.

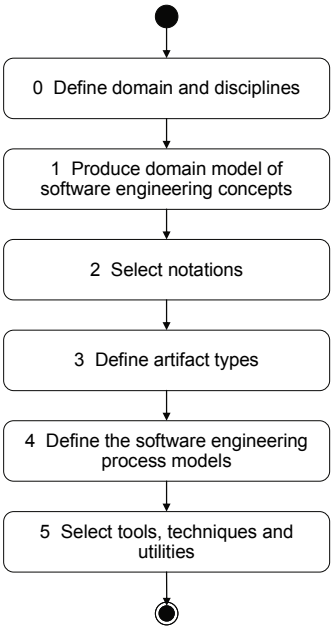


Fig. 12. The fundamental process of the meta-method

2. **Select notations:** In order to represent the software engineering concepts appropriately, notations for their representation are required. Languages, together with possible sub-languages (e.g., UML diagram types) and language elements must be identified (either by newly defining them or by reusing existing notations) and enumerated as candidates. Among them are typically languages for modeling, documentation and implementation of software engineering concepts.

3. **Define artifact types**: Candidate notations, i.e., languages and language elements that have been identified in step 2 are assigned to SE concepts from step 1 according to the properties of the software engineering concepts that need to be expressed. While the domain model of software engineering concepts can be interpreted as the semantic domain of the software engineering method, languages define the syntax for denoting them (notation particularly refers to the concrete syntax). The domain model of software engineering artifacts then defines the semantics of the languages by linking language elements (and particularly their syntactic representation) with SE concepts. Consequently, the given semantics of the proposed candidate notations must be conformant with the semantics imposed by the composition of step 3, and the semantics of each language element shall still be unambiguous. Composition hierarchies in the domain models of SE concepts and artifacts must be compatible.

4. **Define the software engineering process models**: The definition of the software engineering process reifies the definition of a roadmap through the network of development artifacts. Tasks and activities are defined and ordered into workflows that produce the required artifacts in the specified order. Sequential, parallel, iterative, incremental, evolutionary and agile development processes shall be supported. The process model is composed from work models and workflow models.

   We need to define activities for accomplishing tasks of software engineering in a software development project and the process structure, comparable to the work breakdown structure in [OMG08]. The process structure contains activities, milestones and control-flow elements. The definition of tasks and activities can be extended by object flows of used and produced work items (of the given artifact types) and roles that are responsible for executing activities or participate in the activities' execution (the responsibility for an artifact is a structural issue, that is modeled separately; a role model is also provided separately). For defining the process, the following sub-activities need to be performed, which will be explained in more detail with the M1-level examples in Sect. 5.6 and Sect. 5.7:
   a) identify tasks and activities,
   b) define the process structure (workflow) including processes, activities, and milestones,
   c) specify work element structures for tasks, activities and possibly workflow patterns including roles and work products,
   d) define the temporal course using phases and possibly other kinds of time period concepts such as iterations, releases, builds,
   e) describe transformations and constraints.

5. **Select tools, techniques and utilities**: The selection of tools, techniques, and utilities as well as the provision of concepts of use for these tools are required for guiding and simplifying the software engineering work and producing the required software artifacts. Tools, utilities, and guidance are assigned to techniques. Techniques are in turn related with methods to which they belong and/or to activities and tasks in which they are used. Guidance on how to proceed in an activity or task to produce the artifacts of a particular type shall be explicitly

provided, e.g. in the form of guidelines, good and best practices, whitepapers, checklists, templates, examples, or roadmaps. However, even the assignment of languages to software engineering concepts in step 3 can be interpreted as partly associating a technique for the development artifact. Both languages and tools typically have implications on how to produce an artifact. Eventually, tools and utilities are thus related to the activities of the software engineering process model as well. By this, it is shown which activities are supported by tools and utilities and, in turn, which of them are to be used when accomplishing the task of the activity.

Enacting this process in the M1 level of the method engineering domain, we systematically create a software engineering method as an instance of the meta-methods M2 product meta-model.

5.4 Integrating the Views of the Meta-Method

As can be seen from the description of the product and process models of the meta-method in the last two sections, a number of consistency issues arise from the different views on the software engineering method's elements:

- consistency of artifacts as defined in the artifact model and their use as work products in the process,
- conformance of hierarchical composition structures of the domain model of software engineering concepts and the artifact model,
- consistent composition of process structures from activities, obeying all given constraints such as the hierarchical composition of activities and flow relationships (predecessor) between activities, or the linking of roles and work products,
- consistency between work product use in activities and processes and the artifact lifecycle model.

Such issues need to be resolved in the method engineering domain when a software engineering method is defined. In addition to the method's conformance with the meta-methods product meta-model, it is also possible to define additional constraints for software engineering methods as instances of the `Constraint` meta-model class.

Yet, a common meta-model for software engineering methods does not only provide a common language for describing software engineering methods; it can also be used as a standardized and unified reference model for software engineering methods. Different methods can be analyzed, compared and exchanged on this basis.

In addition to the definition of activities and process structures (e.g. depicted as UML activity diagrams holding the different process elements and specifying the workflow of activities), we deploy collaborations in the work model defining the effect of work elements, i.e., tasks and activities, on the artifact structure (which can be interpreted as graph transformation rules that are typed over the artifact model). Such models of the dynamics of a method can for instance be used for reasoning or

analyzing certain properties of a process. These transformation rules are explained in more detail in Sect. 5.7.

Following this general discussion of the meta-method level, we will present the artifact model (result of step 3) and the software process model (step 4) in more detail in the following two sections. They represent instances of the meta-method product and process models (M2) on the method level (M1).

5.5 Defining the Artifact Model of Software Engineering Method

The result of step 3 of our meta-method is a domain model of software engineering artifacts. The objective of this model is to establish a common understanding of the software artifacts that are to be produced in a software development project. This comprises the purpose and meaning of each artifact type and its relevant properties as well as the relationships, associations, dependencies, and generalizations between artifact types. The artifact model is a type model that is used in the role of a meta-model in the software engineering domain and thus instantiated for describing a software system in a software development project (compare Fig. 10). It defines the set of software engineering artifacts that are produced, edited, and used throughout a software development project as part of the software engineering method. It thus constitutes the product model of the software engineering method (M1 in the method engineering domain) and acts as the backbone of a family of methods where it can be combined with dedicated process models, languages and tools.

Figure 13 shows as an example an excerpt from the artifact model for system specifications in the disciplines business modeling and analysis that has been developed together with an industrial partner.

For each artifact type, we can specify an artifact lifecycle model (state model, see Fig. 11) which defines the lifecycle of artifacts that are instances of this artifact type.

5.6 Software Process Modeling in the Software Engineering Method

The workflow structure of the process defines an ordered and hierarchically nested structure of activities and milestones. Ordering relations, i.e., direct predecessor dependencies between elements, are explicitly specified. They may be marked e.g. as dependencies of the appropriate temporal relationship kind (such as typical interval relations startToStart, startToFinish, finishToStart, finishToFinish, compare [OMG08]). Transitive dependencies need to be computed for scheduling the activities. Other properties of the work elements that are relevant for the ordering an execution may be indicated by meta-attributes, such as “hasMultipleOccurences“, "isOptional“, "isRepeatable“, "isOngoing“ and "isEventDriven“ (compare [OMG08]).

The flow of activities can also be represented in UML activity diagrams or other process languages such as BPMN, Petri Nets, etc. The flow diagrams can contain activity elements (as actions) as well as control flow patterns and nodes such as

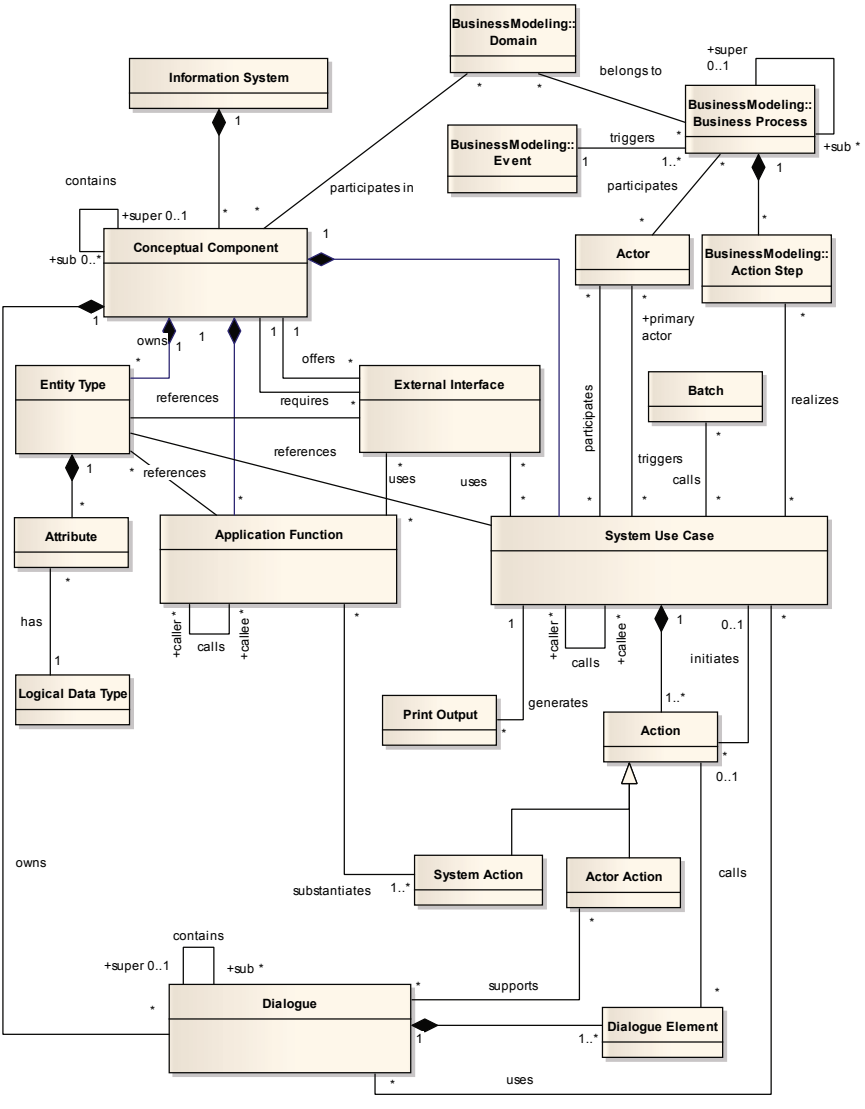


Fig. 13. Excerpt from the artifact model for system specification in the discipline analysis with reference to elements from the discipline business modeling

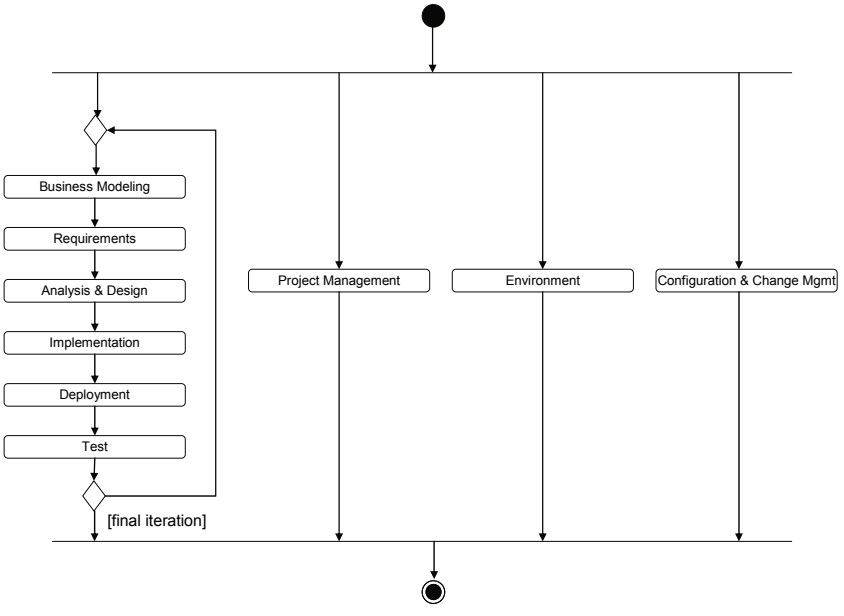


Fig. 14. Flow of high-level activities for the Unified Software Development Process

sequential, alternative and parallel execution, conditional flows and iteration. Thus it is possible to specify different kinds of software engineering processes.

Figure 14 shows an example of an activity diagram depicting the flow of high-level activities according to the disciplines of the Unified Software Development Process [JBR99]. In each iteration of the process, the engineering activities on the left-hand side are executed sequentially. The three activities corresponding to the supporting disciplines on the right-hand side run across the whole development lifecycle.

Constraints may be defined that restrict the possible flow of activity. For example, we can use temporal expressions to define that each activity for specifying a use case must eventually be followed by an activity specifying a test case.

Roles and work products (artifacts) shall be related to activities and milestones. Therefore, they are to be included in the work model. (Within this task, one may finally select from alternative artifact types, by defining the references to the artifact types of the information model.) For activities, we can indicate whether work products are used as parameters of kind „input/output/inoutput“. For milestones, we can indicate which work products are required results that have to be completed for achieving that milestone (responsibility assignment). For all elements reference roles and work products, it may be specified whether they are mandatory or optional by a meta-attribute „isOptional“.

Activity diagrams with object flows can be used to depict the input and output work items of each activity. Work product uses are represented as object flows according to their parameter kind: in, out, inout. One can use ObjectNodes, Pins

ActivityParameterNodes of UML activity diagrams for this purpose. Roles can be integrated by the use of ActivityPartitions (aka. "swimlanes") for assigning activity elements to the corresponding roles, representing the relationship between activity and role use. Figure 14 shows an example how the activity pattern for structuring the use case-model according to the Unified Software Development Process [JBR99] can be represented. This diagram shows the activity together with role use and work product uses as well as the deployment of the tool Enterprise Architect according to step 5 of the meta-method.

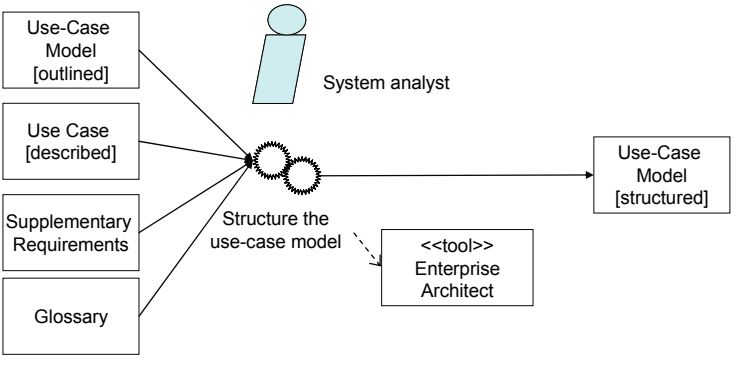


Fig. 15. Activity pattern for the activity "Structure the use-case model" according to [JBR99]

Consequently, the following relationships are represented in such a model:

- a role is responsible for performing an activity,
- a work product is used or produced by an activity as input, output or input-output parameter,
- if a relationship of type performs exists between an activity and a role use and a parameter relationship of kind out or in-out exists between the same activity and a work product use, then there also exists a relationship of type responsible (responsibility assignment) between the work product use and the role use.

5.7 Defining Work of Software Engineering Methods as Transformations

As can be seen from Fig. 15, the representation of the effects of work elements on the artifact model can only be expressed in a limited way by using activity diagrams or composite structures. Even if object flows are represented, they can only make reference to the state of individual objects (such as the states "outlined" and "structured" for the use-case model and the state "described" for use case in Fig. 15). We therefore include transformations that are depicted as UML collaborations (i.e., the structural part of UML 2 interaction diagrams) that are interpreted as graph transformation rules on the type graph of the artifact model (as introduced in [HS01]).

Figure 16 gives an example of such a transformation rule. It states for the activity "Identify system use cases" that for each occurrence of the pattern on the left-hand side in an instance of the artifact model, the structure on the right-hand side must be

produced by the activity. In particular it states that if a business process has an action step that shall be supported by the software system being modeled (property "isManual" = false), then a system use case needs to be included in the system model that realizes this action step and whose primary actor is the same actor who is responsible for executing the action step of the business process. The rule can be interpreted as a visual contract [LSE05] stating pre- and post-conditions of the activity.

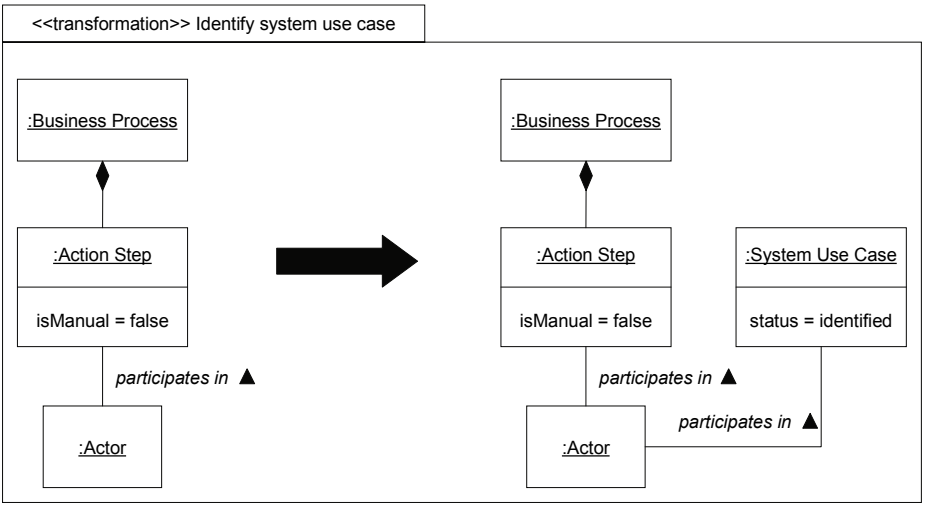


Fig. 16. Transformation rule for activity "Identify system use case"

With this example of a transformation rule that specifies the effect of a work element in a method we conclude our tour of how to develop software engineering methods with MetaME. So far we have focused on how to create new software engineering methods, but we can distinguish between initial development of a software development and its modification. Modification can have different causes such as the need to evolve, tailor, specialize or extend the software engineering method. We will exemplify tailoring scenarios in the next section.

6 Tailorable Software Engineering Methodology

Software engineering methods evolve over time due to changing requirements, new achievements in software engineering – such as new process models and development practices – and experience from software engineering projects. However, each software development project is individual too, at least to some notable degree. This results in at least slightly different requirements for every software development project. Thus, software engineering methods must be tailorable. Such tailoring is a possibility to implement situational method engineering (see Sect. 1). It happens on the M1 level of our meta-modeling hierarchy. We will not discuss changes to the meta-method on the M2 level here, since they should only appear in a controlled

evolution process within the meta-method engineering domain, and are not part of the method engineering domain.

Changes on the M1 level, where the software engineering method is located, can be manifold and will occur rather often. For example, if a project is of limited size and budget, it may not be appropriate to execute all activities in full and to produce all the work products that are defined in the general method. If a method engineer (or a project leader who is responsible for the tailoring of the software engineering method) wants to change the artifact types, their properties or relationships, she has to adapt the software engineering artifacts model. The consequences of such modifications on the network of artifact types can be directly observed from the artifacts model. For example, if a project decides to specify the software system without use cases and to use a combination of business processes, business rules, dialogue specifications and application functions instead, this will have an impact on numerous artifact types that are typically related to use cases.

Changes to the artifacts model will typically also affect the defined activities in the process dimension, since tasks and activities use artifacts from the artifacts model as their input and output objects. If use cases are no longer produced, the work elements producing use cases are no longer required. Work elements that typically depend on the provision of use cases as input need to be altered for using the other supplied artifacts that are available as work products. Maybe even the flow of activity must be altered due to changes in the parameter object types. For example, if dialogue specifications are to be used in an activity instead of use cases, but dialogues were only specified later in the process so far, the activity of specifying dialogues will have to be moved upfront before the dependent activity.

A number of other changes to the software engineering method are also quite common:

- changes of techniques how to produce a work product,
- change of notations for representing the content of artifacts,
- changes of tools, e.g. due to the wish of a customer, to enable tool chaining with another tool, or to interoperate with development partners using a common tool basis,
- addition of roles and responsibilities, and many more.

The advantage of the formal meta-model based approach of method engineering that we have presented in this work is that its parts and their relationships within and among each other are precisely and explicitly modeled. Thus tailoring can as well be executed and described in a systematic way.

7 Conclusion

In this paper, we have presented MetaME, a meta-method for method engineering of software engineering methods. It builds on a four layer meta-model hierarchy which combines the two domains method engineering and software engineering. We described a meta-model as a general product model of method engineering as well as a process for developing software engineering methods that consists of 5+1 steps.

Together they cover the product and the process dimension of the meta-method. The most important steps 3 and 4 of that process, which are concerned with defining an artifact model and software process modeling, respectively, are shown and discussed in detail. There we also introduce the idea of specifying software engineering tasks as transformation rules that are typed over the artifact model. The issue of tailoring in our meta-method for the engineering of software engineering methods is briefly discussed in Sect. 6.

Although method engineering of software engineering methods is not a new domain, there is still work to be done. The integration of the different aspects and views of such a method is not yet complete. Especially the use of constraints and patterns in a meta-model-based approach still needs to be better understood. Furthermore, we have made a first step towards the integration of structural and behavioral meta-modeling of software engineering methods. This integration needs to be continued and evaluated. The integrated meta-modeling architecture underlying the method engineering and software engineering domains appears to be a basis for this.

References

[AK01] Atkinson, C., Kühne, T.: Processes and products in a multi-level metamodeling architecture. Int. J. Softw. Eng. Knowl. Eng. 11(6), 761–783 (2001)

[Bal98] Balzert, H.: Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Spektrum Akademischer Verlag, Heidelberg (1998)

[BG08] Bollain, M., Garbajosa, J.: A metamodel for defining development methodologies. In: Filipe, J., et al. (eds.) ICSOFT/ENASE 2007. CCIS, vol. 22, pp. 414–425. Springer, Heidelberg (2007)

[BKPJ07] Becker, J., Knackstedt, R., Pfeiffer, D., Janiesch, C.: Configurative method engineering - on the applicability of reference modeling mechanisms in method engineering. In: Proc. Americas Conference on Information Systems (AMCIS 2007), paper 56 (2007), http://aisel.aisnet.org/amcis2007/56

[Bri96] Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Inf. Softw. Technol. 38(4), 275–280 (1996)

[ESS08] Engels, G., Sauer, S., Soltenborn, C.: Unternehmensweit verstehen – unternehmensweit entwickeln: Von der Modellierungssprache zur Softwareentwicklungsmethode. Inform. Spektrum 31(5), 451–459 (2008)

[GH08] Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley & Sons, Chichester (2008)

[GMH05] Gonzalez-Perez, C., McBride, T., Henderson-Sellers, B.: A metamodel for assessable software development methodologies. Softw. Qual. J. 13, 195–214 (2005)

[Gut94] Gutzwiller, T.A.: Das CC RIM-Referenzmodell für den Entwurf von betrieblichen, transaktionsorientierten Informationssystemen. Physica-Verlag, Heidelberg (1994)

[Hey95] Heym, W.: Prozeß- und Methoden-Management für Informationssysteme: Überblick und Referenzmodell. Springer, Heidelberg (1995)

[HG05] Henderson-Sellers, B., Gonzalez-Perez, C.: A comparison of four process metamodels and the creation of a new generic standard. Inf. Softw. Technol. 47, 49–65 (2005)

[HR10] Henderson-Sellers, B., Ralyté, J.: Situational method engineering: state-of-the-art review. J. Univers. Comput. Sci. 16(3), 424–478 (2010)

[HS01] Heckel, R., Sauer, S.: Strengthening UML collaboration diagrams by state transformations. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 109–123. Springer, Heidelberg (2001)

[IBM07] IBM Corporation: Rational Unified Process. Version 7.0.1 (2007)

[IEEE90] IEEE: Standard Glossary of Software Engineering Terminology, IEEE Std 610.12, The Institute of Electrical and Electronics Engineers, New York (1990)

[ISO07] ISO: ISO/IEC 24774:2007 Software engineering – metamodel for development methodologies. International Organization for Standardization, Geneva (2007)

[JBR99] Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process: The complete guide to the Unified Process from the original designers. Addison-Wesley, Reading (1999)

[JJM09] Jeusfeld, A., Jarke, M., Mylopoulos, J. (eds.): Metamodeling for method engineering. MIT Press, Cambridge (2009)

[LSE05] Lohmann, M., Sauer, S., Engels, G.: Executable visual contracts. In: 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), pp. 63–70. IEEE Computer Society, Los Alamitos (2005)

[NFK94] Nuseibeh, B., Finkelstein, A., Kramer, J.: Method engineering for multi-perspective software development. Inf. Softw. Technol. 38, 267–274 (1994)

[OMG06] Object Management Group: Meta Object Facility (MOF) Core Specification, Version 2.0 (2006), http://www.omg.org/spec/MOF/2.0/PDF/

[OMG08] Object Management Group: Software & Systems Process Engineering Meta-Model Specification, Version 2.0 (2008), http://www.omg.org/specs/

[OMG09a] Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure, V2.2 (2009), http://www.omg.org/uml/

[OMG09b] Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, V2.2 (2009), http://www.omg.org/uml/

[RBH07] Ralyté, J., Brinkkemper, S., Henderson-Sellers, B. (eds.): Situational Method Engineering: Fundamentals and Experiences, Proc. IFIP WG 8.1 Working Conference. Springer, Boston (2007)

[Rol09] Rolland, C.: Method engineering: towards methods as services. Softw. Process Improv. Pract. 14, 143–164 (2009)

[SB02] Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall, Upper Saddle River (2002)

[SSEB10] Salger, F., Sauer, S., Engels, G., Baumann, A.: Knowledge transfer in global software development – leveraging ontologies, tools, and assessments. In: Proc. 5th Intl. Conf. Global Software Engineering (ICGSE 2010), pp. 336–341. IEEE Computer Society, Los Alamitos (2010)

[Sta09] Stadtler, D.: Eine generische Methode zur unternehmens- bzw. projektspezifischen Festlegung von Vorgehensmodellen zur Entwicklung von Software, Diplomarbeit, Universität Paderborn (2009)

[Str98] Strahringer, S.: Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips. In: Pohl, K., Schürr, A., Vossen, G. (eds.) Modellierung 1998 (1998), http://ceur-ws.org/Vol-9/

[Wie03] Wiegers, K.E.: Software Requirements. Microsoft Press, Redmond (2003)

Techniques for Merging Views of Software Processes*

Josée Tassé[1], Nazim H. Madhavji[2], and Amandeep Azad[3]

[1] University of New Brunswick, P.O. Box 5050, 100 Tucker Park, Saint John, Canada, E2L 4L5, (née Turgeon)
jtasse@unbsj.ca
[2] University of Western Ontario, Middlesex college, London, Canada, N6A 5B7
madhavji@csd.uwo.ca
[3] IBM US, Southfield, Michigan
aazad@us.ibm.com

Abstract. Models of software development processes have many uses such as an aid to understanding, composing, assessing, improving and automating workflows. However, eliciting descriptive models from actual work environments can be quite complex due to multiplicity of roles, activities, artefacts, conditions, distributivity, locations and others. One way to manage this complexity is to elicit, from different sources, partial models (called views) of the subject process and then merge them into a coherent whole. In this paper, we describe "view-merging" algorithms, which form a core part of a view-based model elicitation system. The algorithms aid in identifying overlaps and inconsistencies and in presenting possible resolutions which, by interacting with the model elicitor, leads towards an incrementally built, unified, coherent process model. These algorithms have been implemented in a system called V-elicit, which has been validated empirically.

Keywords: software process models, views, view merging, component matching, inconsistencies.

1 Introduction

Software process improvement is often performed by first creating a model of the current process and then analysing it to find improvement opportunities (see, e.g., [1, 9, 29]. The task of gathering information about the process, modelling this information, analysing the model for consistency and completeness, and validating it, is called process model elicitation [26].

For such a task, one may need to use multiple sources of information (different people, documents, observations, etc.) in order to obtain a comprehensive picture of the process being modelled. The reason is that, quite often, no single person in a large

* This work was supported in part by the Natural Science and Engineering Research Council of Canada (NSERC).

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 441–472, 2010.

organisation has innate knowledge about the entire process to the extent of the details needed to make improvements [11, 29, 38]. From our own experience (see details in Section 7 below), the percentage of overlap found across two different people's description of the same process was between 4% and 54%.

However, a wicked problem caused by using multiple sources of information is that the descriptive fragments from different sources (called *views*) may have conflicts among them [20, 29, 38] with respect to details such as: types and instances of the activities and artefacts captured, conditions associated with model elements, timing information, the semantics of similarly named elements, differing terminology used for the same semantics, presence or not of certain relationships among the defined entities, structural differences in the way development tasks and artefacts are grouped to form the model (or view) hierarchies, the level of details represented, and others. These conflicts are there due to stakeholders' personal perceptions and level of understanding of the entities, relationships, conditions, timing, resources and other values involved in the software processes being conducted in a project. Also, in many cases such conflicts are inevitable when there is no single defined and followed process organisation-wide, e.g., organisations low on process maturity scale. Even once mature organisation can relegate to this problem due to corporate mergers and acquisitions, or market impact on survivability and personnel attrition, before it has uplifted itself again to a more mature state. Not only it is recognized that different people may see the process in different – conflicting – ways, such information (i.e., the inconsistencies found) is even used in [1, 7] as a starting point for identifying improvement opportunities.

We take a position that for many investigative or improvement efforts, there is a need for a comprehensive and consistent descriptive process model so that it can be trusted for making subsequent decisions. Failing to adequately understand the descriptive process can lead to problems in defining the requirements for process improvement which, in turn, can result in poorly changed processes or in improvement project failures. Quoting Humphrey [18], "if you don't know where you are, a map won't help". This is supported by Hardgrave and Armstrong [16]: simply trying to implement the CMM key process areas in Level 2 lead to an improvement failure. However, once they put their emphasis on obtaining the descriptive process model and using it to identify improvement opportunities, they succeeded in attaining CMM Level 2 certification.

Building a coherent model from multiple views is non-trivial because it requires that these views are merged in a systematic manner. This task includes identification of the overlapping components across the views, and detection and resolution of the described types of inconsistencies. In the past, view merging was mostly carried out manually [11, 20, 29]. In a large project, there are many disadvantages to such an approach however. For example, the size and complexity of the processes can be a factor in overlooking some overlaps and inconsistencies across different views. The amount, structure and semantics of the information that the elicitor has to deal with from each view (easily hundreds of items) are difficult to manage. Yet, it is important to examine all the views when identifying overlaps and inconsistencies.

In the case study described in [13], a team of subjects used a manual view-based approach for building a requirements model from multiple views. This team failed to build a merged model because of the difficulty in reconciling conflicts manually.

Also, the need for tool support for view-merging was evident from an experiment we had conducted [37] on comparing view-based approaches against traditional ones. The subjects using a traditional approach to process modelling dealt with multiple views by first modelling what they considered to be the most central view of the process, and then by incrementally adding information from the other views. In our analysis of the resultant models, we could trace back most of the completeness or consistency problems to the traditional approach used (i.e., the information from the central view was modelled quite well compared to the information from other views). These problems suggest that an appropriate tool support is needed for identifying overlaps, detecting inconsistencies and for merging different views into a coherent whole. To our knowledge, such a tool support is not currently available – at least not for the types of models to be dealt with in the software process modeling area (see Section 6 where we discuss tools for merging models in domains such as ontology merging and version control management).

To overcome this shortfall, in [33, 36], we describe a view-based elicitation approach with tool support (the V-elicit system). In this approach, the elicitation effort is first planned (through identification of the purpose and scope of the model to be elicited, and the possible sources of information), and views are then elicited from disparate sources and each is checked internally for consistency and completeness. These views are then analysed for overlaps and inconsistencies across them. While these are being resolved, a merged model is synthesised automatically in the background.

Since the publication of these papers [33, 36], we have improved the internal algorithms used, and validated these algorithms further. What is new in this paper is that we have augmented our previous work on view-based elicitation, by giving additional algorithmic details. So the focus and novelty here is on the description of two algorithms used for merging the views: (i) component matching and (ii) inconsistency management. The details on other elicitation steps (e.g., planning, view elicitation, and consistency checking of the individual views), described in earlier papers [33, 36] are not repeated here.

The *component matching* algorithm is used to identify overlaps between two given views. The *inconsistency management* algorithm detects inconsistencies across views, categorising them, and presenting the possible solutions of the inconsistency to the elicitor. In order to help select the right solution, appropriate context information (such as the solution implemented in each view) is provided as well. Together, these two algorithms (called "merging algorithms") aim at synthesising a unified, coherent merged model. Their automation, as described in this paper, makes view-based elicitation practical by ensuring that all the identified overlaps and inconsistencies are considered, and by reducing the amount of work that the elicitor has to perform manually.

This paper is organised as follows: Section 2 gives an overview of our approach through an example. Sections 3 and 4 describe the algorithms used. Discussions on various issues related to the modelling notation and the scalability of the approach follows in Section 5. Section 6 presents related work and comparisons. Section 7 describes the validation performed. Finally Section 8 concludes the paper.

2 View Merging – The Basic Idea

Prior to delving into the two core algorithms (Sections 3 and 4), we first consider a simple, yet illustrative, example of merging so as to obtain a feel for the issues involved in merging a set of disparate views of a given process.

Consider, for example, two[1] views describing a document review process from different angles: Bob's view (Figure 1) as a reviewer and Peter's view (Figure 2) as a document writer. For ease of understanding, we limit the views to activity decomposition and dataflows. However, in a large or complex process, there could as well exist several different views for a particular sub-process, and possibly other kinds of information such as agents' roles, timing, entry/exit criteria, etc. Let us also assume that, as a prerequisite, each of the views is internally consistent and complete[2] before entering the view merging process.

The views are specified as entity-relationship diagrams[3], with user-defined types assigned to entities (e.g., activity, artefact, role) and to relationships (e.g., activity is-composed-of activity, activity produces artefact, activity is-performed-by role). Entities can also have attributes attached to them (e.g., duration and frequency for activities, reference number and time of final approval for artefacts).

The first step in the merging process is *component matching*, to detect overlapping elements across the views. This can detect, for example, that "do_first_version" in Bob's view and "deliverable_production" in Peter's view refer to the same activity. This task is not obvious due to terminology differences and due to the fact that the elements may not mean exactly the same thing (e.g., whether or not the activity of the production includes the activity of modification of deliverables). Thus, we need to examine the descriptions of the entities (i.e., their relationships with other entities and their attributes), and the terminology used in order to match them.

Our approach is to compute a similarity score ([0..1]) for each pair of entities in each pair of views, based on the information related to these entities (i.e., name, relationships and attributes). For our working example, the scores computed by the algorithm are indicated in Table 1. From these, the matching activities are identified as the ones above a certain threshold provided by the elicitor (e.g., 0.5)[4]. The potential matches are indicated by a score in bold in Table 1.

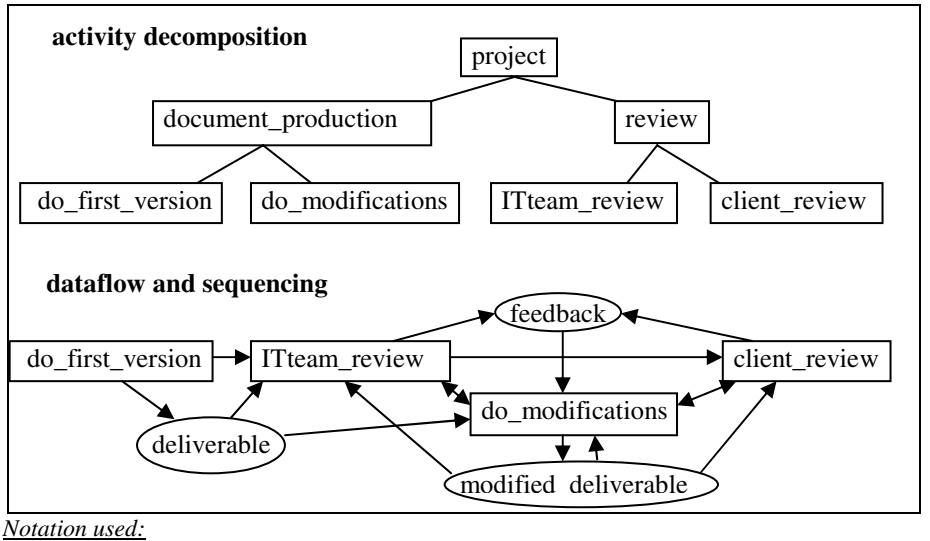
---

[1] The V-elicit system [33] can handle multiple views and has undergone validation using industrial case studies.

[2] These quality factors can be evaluated through a set of rules based on the syntax and static-semantics of the modeling language used (e.g., for completeness: ensure that each defined activity has at least one input and one output associated with it; for consistency: ensure that dependencies among artifacts do not form a cycle). The V-elicit system has such rules for intra-view analysis, which are described in [33, 35] and are not repeated here.

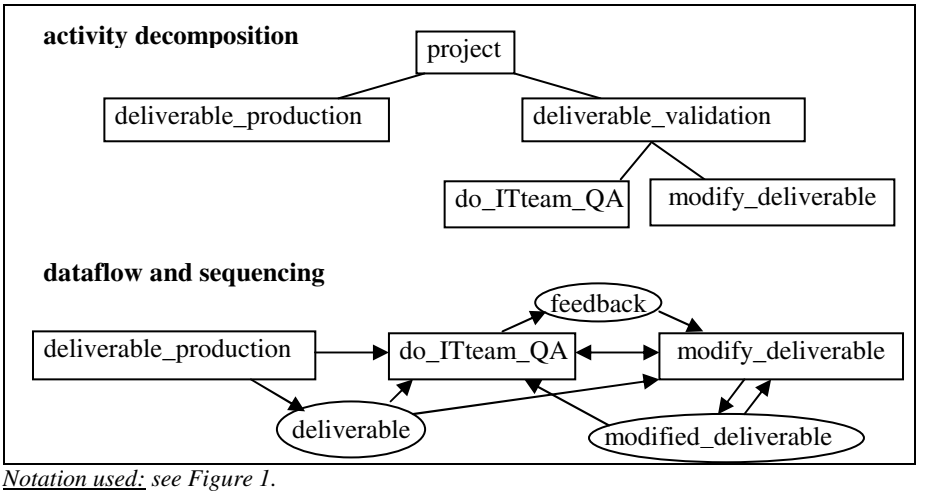
[3] We discuss later, in Section 5, that this choice does not limit the generality of the approach.

[4] This threshold would typically come from past experience. Our own experience suggests that 0.5 is a reasonable value, which typically discriminates between right and wrong matches, while limiting the set of matching pairs presented to the elicitor. Since the values computed are always available for analysis, this threshold is just used to ease the initial analysis of the matches found.



Notation used:
The activity decomposition (top part of the figure) shows the activities involved, at different levels of details, and the composition relationships among them. The dataflow and sequencing aspect (bottom part of the figure) is a combined one, showing activities in rectangles and related artefacts in ellipses. The dataflow relationships are the ones involving artefacts, and they represent the relationships of type "activity produces artefact" and "artefact is-consumed-by activity". The sequencing between activities is shown in two different types of relationships among activities: "activity precedes activity" is shown as regular arrows, and "activity iterates-with activity" is shown as arrows with two arrowheads.

Fig. 1. Bob's view (reviewer)



Notation used: see Figure 1.

Fig. 2. Peter's view (document writer)

Table 1. Similarity scores between activities in Bob's and Peter's view

With a threshold set to 0.5, potential matches of activities between the two views (those above the threshold) are highlighted to the elicitor for analysis. Note that the matches are not always one-to-one.

		Peter			
		deliverable production	deliverable validation	do_ITteam _ QA	modify_ deliverable
Bob	document_ production	0.229	0.298	0	0.298
	do_first_version	**0.620**	0	0	0
	do_modifications	0	0.298	0	**0.705**
	review	0	0.298	**0.593**	0
	ITteam_review	0	0.298	**0.714**	0
	client_review	0	0	0.376	0

As one can see, there could be more than one potential match presented, associated with a single entity (e.g., with "do_ITteam_QA" here), and the elicitor should be involved in this case to choose the right match (typically based on his/her understanding of the elicited information). Such situation may already indicate an inconsistency across views, which can be dealt with at this point or later during the inconsistency management phase. In the case the elicitor prefers to make modifications at this point, the tool provides suggestions to fix the problem in one view, such as adding a more abstract or more detailed entity, or changing the position of an entity in the decomposition view. The danger in handling such case at this point is that a decision is taken without a proper analysis of the impact in all other views (which is handled in the next step – inconsistency management). In our example, assuming that the elicitor chooses to match "do_ITteam_QA" with "ITteam_review", then Table 2 shows the result of this component matching step of the merging process i.e., the list of matches identified. The top-level activity ("project") is not considered here in the matches, as it is supposed to be the same for all the views since it represents the entire process modelled.

Table 2. Final matches between two given views

Bob	Peter
- do_first_version	- deliverable_production
- do_modifications	- modify_deliverable
- ITteam_review	- do_ITteam_QA

It should be noted that at this point, the elicitor's involvement should not be restricted to analyzing cases where more than one match was found, but also to review the matches found and accept them or reject them, possibly proposing better ones. As described in [33], Table 1 above is not presented "as is" to the user, but the user is presented with the probable matches (in a format similar to Table 2) and help is provided for making the proper selection of matches – same as in the special case above

where multiple matches to a single entity are found. At first, such a check might seem time consuming. However, this is greatly reduced by the use of the calculated similarity scores: in fact, the elicitor just needs to consider the entities with higher scores and the other entities in close proximity in order to perform such a check. Thus, not all possible pairs of entities need to be checked, as would be required in a completely manual approach.

Knowing where the overlaps are in the views, we can now attempt to detect inconsistencies across the views, if any, and resolve them (i.e., inconsistency management). The first key step is to ensure that all elements are captured once, and only once, in the merged model, with a proper "definition" of each of them in terms of the set of sub-entities composing them in the entity decomposition[5] (i.e., eliminating all inconsistencies related to the entity decomposition). In our example, we have the following inconsistencies:

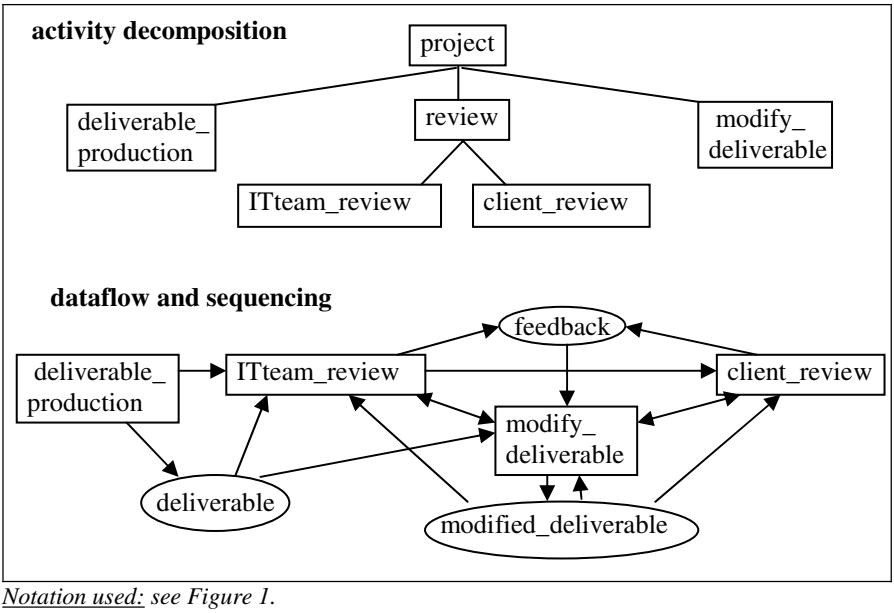
1. "do_first_version" and "deliverable_production" are at different levels in the model decomposition
2. "client_review" is not in Peter's view
3. "do_modifications" (or "modify_deliverable") is grouped with the document production activities in Bob's view, but grouped with the review activities in Peter's view.

For this task, the V-elicit tool works in a top-down fashion, presenting each inconsistency as they are discovered, with possible solutions (see [33] for an example with screen shots). The elicitor simply needs to select the correct alternative from the solutions provided. For example, in the case of inconsistency #1 listed above, the elicitor would have to decide whether or not to keep the "document_production" activity in the merged model and make appropriate command choices in the V-elicit tool. The list of all the views implementing each of these two solutions is also provided (*not only the two views where the inconsistency is originally found*), so that the elicitor can perform a better analysis of which solution is best. Once the set of entities is fixed, inconsistencies related to differences in names and attribute values, as well as non-decomposition relationships missing in some views, can be handled easily (still with each of them presented together with contextual information in order to ease the selection of the proper alternative).

The final result of applying these two mechanisms (component matching and inconsistency management) on the given set of views is one merged model. For our example, one possible merged model is as shown in Figure 3. This final model should also be checked for consistency. Especially, we should check that the final model does not contain unconnected parts. This would indicate that the views were describing adjacent sub-processes (or orthogonal views) without specifying the interface between them. In such circumstances, additional information from the related views has to be elicited to solve this problem. Of course, it would be preferable to prevent such a situation by properly planning the scope of each view prior to their elicitation (handled in the planning step of V-elicit [35]).

[5] The decomposition (or hierarchical) structure is chosen to assess this quality attribute because this view lends itself to the analysis of the existence of elements in a given process model and how these elements are related to other elements in forming the more abstract elements. Other views, e.g., data-flow, role-activity, etc., do not readily serve this purpose.

The component matching algorithm and the inconsistency management algorithm are described in Section 3 and Section 4 respectively.



Notation used: see Figure 1.

Fig. 3. Final model (merging of Bob's view and Peter's view)

3 Component Matching Algorithm

As we saw in the example, in order to merge the views we need to identify the information (e.g., entities and relationships) that overlaps across the views. This is accomplished by computing similarity scores for all the pairs of entities from different views. The similarity score is a number between 0 and 1, where 1 represents entities that are exactly the same.

The general idea in computing a similarity score between two given entities is to compare the elements related to these entities (i.e., name, relationships and attributes). For example, if we look at Bob's activity "do_first_version" (Figure 1), there are four related elements: the name of the entity, and the relationships "document_production is-composed-of do_first_version", "do_first_version produces deliverable", and "do_first_version precedes ITteam_review". When comparing this activity to Peter's activity "deliverable_production" (Figure 2), we need to check whether the latter activity also contains these four related elements.

It is straightforward to compare entity names and related attributes, but not relationships because they involve other entities, which may not have been matched as yet. For example, how can we know that Bob's relationship "do_first_version precedes ITteam_review" is the same as Peter's relationship "deliverable_production

precedes do_ITteam_QA", if we have not yet compared the activities "IT-team_review" and "do_ITteam_QA"?

The approach we have taken is to match the entities one type at a time (i.e., say, artefacts first and then activities), and to use the results of the previous iterations for comparing the related components of the entities in the current comparison. This choice (order in which the entity types – artefacts or activities – are handled) is made by the elicitor, considering such factors as the expected terminology similarity between similar elements (e.g., role names would probably be the same across views if they are commonly-known people's title in the organisation), and the potential use of attribute values as a way to identify common elements (e.g., when the main artefacts are kept under configuration management, with an ID – as attribute – associated with them).

In our example, artefacts are matched first, using solely the name similarity. Then, activities are matched using name similarity, relationships to the matched artefacts, and relationships with other activities. Since the first case (artefact matching) is rather obvious, we focus below on the, structurally more complex, activity matching. The algorithm used for activities, however, can also be used for artefacts (or any other entity type) when terminology differences are expected.

In order to be able to use as much information as possible in the similarity computation, *two* passes are used. The *first* pass computes a partial score using information that we are certain about (name, attributes, and relationships to entities already matched). This partial score can provide some indication of the similarity between the entities of the same type (although not perfectly). The computed partial scores can then be used, in the second pass, to assess the similarity between the *relationships* involving these entities. In essence, the *second* pass uses the indicative information from the first pass to refine the scores resultant from the first pass.

The two-pass feature is particularly useful, for example, when matching artefacts when no other entities have yet been matched: deliverables are usually simple to match because they generally have a title known to its users (especially if configuration management is used), but intermediate process artefacts are not as simple. Thus, by using artefact dependency relationships in the similarity computation, it becomes possible to find such non-obvious matches.

The algorithms used for each of these two passes are described in Sections 3.1 and 3.2 respectively. A general discussion of the algorithms is then provided in Section 3.3.

3.1 First Pass Score Computation

The first pass (resulting in one score per pair of entities compared) involves one *sub-pass* per type of information used in assessing the similarity. For example, assuming that a given pair of activities (A and B) has related elements as "name" and "relationship with artefacts" then we would need two sub-passes: one for the name similarity, and one for the similarities in the relationships with artefacts. In general, there could be other sub-passes corresponding to other elements such as attributes (which are all handled in a single sub-pass), or relationships involving roles of agents in the process.

Regardless, for each sub-pass, a similarity score ([0..1]) is computed, stored, and then combined using weights. For our example (Figures 1 and 2), two sub-passes are required for matching activities, which are then combined as follows:

$$[6]\text{FirstPassScore}(A,B) = W_1 * \text{NameScore}(A,B) + W_2 * \text{ArtefactScore}(A,B)$$

where NameScore(A,B) is the score computed between the activities A and B in the sub-passes related to name similarity; ArtefactScore(A,B) is the score denoting the similarity in relationships with the artefacts of activities A and B; and W_1 and W_2 are the respective weights applied.

The weights used, which can be any positive number, indicate the relative trust we have in the information used in each sub-pass, in terms of our confidence that the type of information used in that sub-pass is a good discriminator for identifying similar elements. For example, if the elicitor feels that the terminology used (entity names) could be different for the same entities in different views, but would rarely be overloaded (i.e., giving a similar name to two different entities), then a higher weight can be applied to the pass related to names. In the case of relationships to artefacts, if in general the artefacts seem to be linked to very few (e.g., one or two) activities, a higher weight can be used than when the artefacts seem to be linked to many activities: in the latter case, the chances that two activities having similar links in two different views are in fact the same are reduced.

The computation of the score in each sub-pass is done in a similar way: two lists of related elements (one for A and one for B) are generated and compared. In the case of *relationship-based* sub-passes, the list contains all other entities linked to the entity being assessed, together with the type of relationship involved (e.g., for "client_review" in Bob's view (Figure 1), the generated list would be {modified_deliverable (consumes relationship), feedback (produces relationship)}). In the case of the *name* sub-pass, the list represents the set of words in the entity's name (e.g., for "client_review" in Bob's view, the generated list would be {"client", "review"}). Comma, hyphen and underscore are the characters used as word separators. The common prepositions and conjunctions in the English language (such as "the", "of", "for", "from", etc.) are disregarded. Furthermore, suffixes (such as "s", "es", "y", "ies", "ation", etc.) are removed from the words to obtain their roots (so that, for example, "modify" and "modifies" are considered the same). Both the prepositions and the suffixes are handled using a simple list of keywords to be checked.

For computing the similarity between the two lists generated, we use the formula in Figure 4. It has been adapted from the "Tripartite Similarity Index" [34], to meet our specific needs. Basically, the first cost function (U) concentrates on the similarity in the size of the two lists, and the second one (R) considers the proportion of similar elements in the two lists. Those were important requirements in our choice of similarity index, due to the heavy use of decomposition relationships in the views. For example, the similarity index had to clearly differentiate between a low-level activity producing a single document, and another activity (ancestor of the first one) producing the same document as well as five others. This is why the size of the lists matters here. A thorough analysis of 20 other similarity indices described in [34] shows that

[6] Remark: FirstPassScore(A,B) = FirstPassScore(B,A). This also holds for all other scores computed.

they had a problem of sensitivity to the variation in the size of the lists, in particular when the two lists compared were of different sizes. The examples used for analysis could arise in our domain. Thus, we rejected these indices in favour of Bipartite Similarity Index. Other requirements satisfied by this chosen index are the linearity and invariance properties (discussed later in section 3.3).

Let: a = number of elements common in the two lists
b = number of elements in the first list that are not in the second list
c = number of elements in the second list that are not in the first list

The two cost functions, which use these contributing factors, are as follows:

$$U = log_2 (1 + (min (b,c) + a) / (max(b,c) + a))$$

$$R = log_2 (1 + a / (a + b)) * log_2 (1 + a / (a + c))$$

And the resulting similarity score is:

$$score = \sqrt{U * R}$$

Fig. 4. Bipartite Similarity Index for comparing two lists

3.2 Second Pass Score Computation

Once all first pass scores are computed for each pair of activities between given two views, these scores are improved in the second pass. But first, an initial screening of the computed results is performed: all scores below a certain threshold[7] are disregarded by resetting them to zero. From our experience, this is necessary for reducing the number of false matches found.

In the calculation of the second pass score, an activity score[8] is first computed using relationships to other activities. The computation needs to consider the position of the activity nodes (i.e., A and B in ActivityScore(A,B)) in the respective tree decompositions. The three possible cases are:

(a) Leaf / Leaf activity comparison (i.e., both A and B are leaves)
(b) Leaf / Non-leaf activity comparison (i.e., A or B is a leaf, the other is not)
(c) Non-leaf / Non-leaf activity comparison (i.e., both A and B are not leaves)

Figures 5, 6 and 7 show the algorithms used for these three cases respectively. Notice that for a given pair of entities A and B, only one of these situation would apply, and thus the computation of ActivityScore(A,B) would use only one of the 3 approaches. The computation of all scores is performed bottom-up (as upward traversal of the activity decomposition trees in each view), and all the scores are stored. All leaf/leaf comparisons are performed first, followed by all leaf/non-leaf cases, and finally all non-leaf/non-leaf cases.

[7] The threshold value, provided by the elicitor, is based on experience. This value shall represent approximately a value that could be obtained by chance.

[8] Notice here that we are describing an "activity score", but it is actually a score based on the type of entity currently being matched. For example, if we were in the second pass of matching artefacts, then we would use an artefact score here, using relationships among artefacts.

Assuming that the entities compared are A from the first view, and B from the second view, the steps are:

1. R_A = list of relationships starting from entity A
   R_B = list of relationships starting from entity B
   Remark: if a relationship arrives to A (or B), the relationship is inversed first, including its type (e.g., "X precedes A" becomes "A follows X")
2. T_A = list of relationship types used in R_A
   T_B = list of relationship types used in R_B
3. $T = T_A \cap T_B$
4. Initialize score to zero
5. For all element t in T
   5.1. S_A = list of entities (other than A) involved in the relationships of type t from R_A
   S_B = same as S_A, for B and R_B
   5.2. S_{max} = longest list between S_A and S_B
   S_{min} = shortest list between S_A and S_B
   (arbitrarily assigned if same length)
   5.3. For each element e in S_{max}
   Add to score the highest FirstPassScore between e and any element of S_{min}
6. If (parent of A ≠ root) and (parent of B ≠ root)
   Add FirstPassScore(parent of A, parent of B) to score

Fig. 5. Case (a) -- Leaf/Leaf comparison algorithm (activity score)

Assuming that the entities compared are A (leaf) from the first view, and B (non-leaf) from the second view, the steps are:

1. Calculate the average of the Activity pass scores between A and the children activities of B.
2. If both the parent of activity A and the parent of activity B are not the root activity, add the First pass score between their parents to get the final Activity pass score between the two.

Fig. 6. Case (b) Leaf/Non-leaf comparison algorithm (activity score)

The activity scores are then combined with the result of the first pass comparisons using the following formula:

$$\text{SecondPassScore}(A,B) = \text{FirstPassScore}(A,B) + W_i * \text{ActivityScore}(A,B)$$

where FirstPassScore(A,B) is the score computed using the formula in Section 3.1; ActivityScore(A,B) is the score computed using the algorithms in Figures 5, 6, and 7; and W_i is the weight attached to the activity score (defined in a similar way as for the weights used in the computation of the FirstPassScore – see section 3.1).

Assuming that the entities compared are A from the first view, and B from the second view, the steps are:

1. Extract two sets S1 and S2, containing the children activities of A and B respectively.
2. Initialize set $S_{final} \leftarrow \phi$
3. For each activity a in S1
   3.1. Finds the activity b in S2 which gives the best activity score with a
   3.2. Add Activity_pass_score(a,b) to S_{final}
4. Repeat Step 3 with S2 instead, storing results in S_{final}
5. Activity_pass_score(A,B) = average of scores in S_{final}
6. If both the parent of activity A and the parent of activity B are not the root activity, add the First pass score between their parents to get the final Activity pass score between the two.

Fig. 7. Case (c) -- Non-leaf/Non-leaf comparison algorithm (activity score)

But since the activity score is not in the same range as the sub-pass scores from the first pass[9], it has to be scaled down to the regular [0..1] range. This is performed by β–conversion (see Figure 8). The final score (SecondPassScore) also has a similar problem of range, and the β–conversion is used as well in the last step of computing the final similarity score.

Let $\beta = 1 / D$ where D is the average of all non-zero values to be converted.
Then for each value v to be converted to the [0..1] range:

$$\text{Scaled-down value} = (\beta * v) / (\beta * v + 1)$$

Formula used to convert a set of values in no particular range (SecondPassScore) into the range [0..1], as described in [31].

Fig. 8. Formula for β–conversion

The final similarity scores provided in Table 1 were computed using the following values (chosen from our experience with real-world process fragments used when developing this algorithm)[10]:

W_1=1.25, W_2=1, W_i=1, cut-off=1 (end of first pass)

Such calculation is performed for each entity type used in the pair of views under consideration.

3.3 Summary and Discussion of Component Matching

In summary, there are two computational passes for matching the components across the views. The first one uses information that we are certain about for assessing the

[9] The activity score is basically the sum of a number of first pass scores, one per relationship used.
[10] An empirical analysis for those weights is provided at the end of Section 7.2.

similarity (one sub-pass per type of information used), and the second one improves the score obtained by looking at the similarity of relationships across the entities of the same type (activities here). The scores computed are first used to identify the truly similar entities (matches), in a semi-automatic way by providing a list of probable matches to the elicitor and then letting him/her analyse the scores computed to select the best matches. Then, once the matches are found for a given type (e.g., artefacts), these can be used in the calculation of the similarity scores between entities of another type.

In the example provided, the artefacts were matched simply by associating artefacts with the same name (for simplicity). In reality, there would have been some similar artefacts with different names (not the commonly-known deliverables, but the temporary artefacts used such as memos or drafts). In this case, the similarity computation could have used names and attributes (title, time of creation or modification, etc., if such information is available) in the first pass, and then dependency relationships across artefacts in the second pass. Activities, which are not matched at the time of the artefact matching process, are not considered at all in the calculation. It should be noted that attribute values do not need to be perfectly equal to each other in order to be considered the same: a percentage of error can be accommodated, as specified by the elicitor (e.g., in a project lasting few months, one task of 4 days in duration is considered of approximately the same length as one lasting one week).

Two variants of the algorithms presented above are possible. First, for any of the sub-pass scores, the elicitor may decide to allow the score to take the values 0 or 1 only (i.e., considering a similarity only when the two lists compared are exactly the same). This could be useful for example when considering relationships to roles, where only few roles are used and many activities are performed by the same roles: the discriminating factor here would be to have exactly the same set of people (team) performing the activity.

The second variant is to perform the algorithms above iteratively: once some matches are found (using the first and second passes as described in Sections 3.1 and 3.2) and approved by the elicitor, the second pass scores are re-computed, using a score of 1 instead of the partial (first pass) score for entities matched during previous iterations. This helps in finding the more obscure matches based on the obvious ones. Note that this approach could also be applied in the case that some initial matches were found manually – and easily – by the elicitor. There are no fixed set of criteria for choosing the right number of matches at a given iteration, but as a rule of thumb, one should choose the matches corresponding to the top scores which are considered to be distinctively higher than the other scores. During our investigation, we never used more than two iterations, as the later iterations would include matches of entities that are not exactly the same[11], which would then render the recalculated scores misleading when choosing other matches (in the subsequent iterations).

[11] Recall that, as explained in Section 2, the idea of matches is not to detect only entities that are perfectly similar. A match can be on entities that are "almost" the same. For example, in one view the document validation phase could include the modifications to the document, while the second view could see the validation phase as only the review process. In such a case, the two validation phases could still be considered as matched. The difference in their definition can either be resolved while finding the matches (component matching algorithm) or while dealing with the inconsistencies across views (inconsistency management algorithm, as described in Section 4).

The chances of finding the right matches could possibly be improved through the use a suitable ontology[12] (or any other document providing standardized terms and definitions, e.g., the ISO/IEC 12207 standard) while modeling views separately, whenever possible. For example, if a given activity modeled in a view is declared to be representing a certain term in a given ontology, then an appropriate link could be specified in an attribute of this activity. A high weight could then be assigned to such an attribute in the calculation of the First Pass Score, thus easing the matching of entities across views which have a common link in the ontology. It should be noted though that it may not be possible to always designate such a link (i.e., not all activities in a given view would definitively have a corresponding term in the ontology). Even the ISO/IEC 12207 standard, with its large set of development activities described, indicates that such a set should be tailored to the particular needs of an organization or a particular project.

The algorithms described in Sections 3.1 and 3.2 have been developed through a thorough analysis of real-world examples (different from the ones used for validation) with many approaches tried in order to find the best ones. The validation of these algorithms includes their use in eliciting three different processes from various industrial settings (as described in Section 7). The Bipartite Similarity Index used for comparing lists (see Figure 4) has also been tested for invariance and linearity[13]. Overall, we ensured that the various scores returned (overall ones or individual ones returned by the Bipartite Similarity Index) intuitively made sense in various typical scenarios we looked at. Our experience shows however that these algorithms should be used as a heuristic only, for identifying the overlapping elements of the model. They may sometimes fail to identify some of the matches, especially when there is not much information available to be used in the similarity score calculation. However, in the examples and industry-scale models tested in this investigation, the results given by the system were satisfactory, with approximately 80% of the matches correctly found, and the computed scores (especially those lower than the threshold but still high enough to indicate some possible similarity) helping to identify the remaining ones.

The result of the matching algorithm presented above (including the final analysis of the computed scores by the elicitor in order to confirm all matches) is the list of all pairs of components, between two views, that are considered similar (for example, see Table 2). However, in any given elicitation exercise, there can be more than two views. In such a case, each pair of views should independently go through the same process of matching components between the two views (i.e., for N views, there would be a need to perform N*(N-1) times the matching algorithm above). This

[12] Note that we are not aware of the existence of any ontology that covers the whole set of software development activities, and which is universally adhered to.

[13] Let BSI(a,b,c) be the score computed using the formula in Figure 4 (with a, b, and c – the "contributing factors" – as described in Figure 4). Invariance means that the relative magnitude of the contributing factors would not influence the result of the score computation (i.e., for any combination of contributing factors (a,b,c) and any arbitrary factor n, BSI(a,b,c)=BSI(n*a, n*b, n*c)). Linearity means that for an equal amount of change in the combination of the contributing factors (a,b,c), there should be a change in the value of the score computed (i.e., BSI(a+2,b-2,c-2) – BSI(a+1,b-1,c-1) = BSI(a+1,b-1,c-1) – BSI(a,b,c)). The result of these tests shows that the BSI (Bipartite Similarity Index) is perfectly invariant and almost linear. These are important properties for similarity indices.

should be done before performing the next algorithm ("inconsistency management algorithm"), as this one identifies the inconsistencies across all views at once, and requires the list of matches found.

4 Inconsistency Management Algorithm

Having identified the matching entities and relationships[14] across a set of views, in the second phase we attempt to identify and resolve any inconsistencies across the views. For each inconsistency detected, a separate window with a set of possible resolutions is presented to the elicitor to obtain his/her choice.

Four categories of inconsistencies are handled in turn corresponding to the following: (i) entity decomposition (including structural differences and differences in the details provided), (ii) entity names, (iii) values of the attributes, and (iv) relationships to other entities. The first category identifies the entities that should be represented in the final model. It should be performed first because it has an impact on the other categories. Following this, other categories can be performed in any order.

When detecting and resolving the inconsistencies related to entity decomposition, the V-elicit system works with one entity type at a time (e.g., activities first, then artefacts - the order here is not important). Within that entity type, the ordering of the inconsistencies dealt with is top-down, from the root to the leaves of the decomposition tree, in a recursive manner. In the case that there is no single root, a temporary one is added. As the inconsistencies are resolved, the final model is built concurrently. A copy of each view is also maintained, modified as the inconsistencies are resolved, to reflect the previous decisions taken regarding the solutions chosen. This is useful for avoiding detection of inconsistencies that no longer exist.

The first step in this algorithm is to make sure that all the views contain the same root. If that is not the case, an inconsistency is flagged, and a common root should be identified by the elicitor. This common root is added to the final model, and to all the copies of the views. In our working example (Figures 1 and 2), this is not necessary for the activities because both views have the same activity "project" as the root, but the artefacts, which are not structured, would need one.

We then recursively detect and solve the inconsistencies related to each node, starting at the common root. Before recursively checking the children of a node, we have to make sure that in each view we have the same children for this node, and that the subtrees under these children contain the same matched entities. Once this is done, the children of a node are all copied in the final model being built, and in the copies of the views not containing it, before recursion. The traversal through each entity is thus made on all views at the same time.

Again in our working example, starting at the "project" activity, a first problem detected would be that the activities are not grouped in the same way: the "do_modifications" (or "modify_deliverable") activity is not under the same subtree in both views. Upon provision of the possible resolutions by the system, the elicitor can then choose to keep it under any of the two subtrees, or as a direct child of the

[14] Two relationships are matched if they have the same type (e.g., an activity producing an artefact) and if the two entities involved in the relationship are also matched in the other view.

"project" activity. Assuming that the elicitor chooses the last option, then the modification activity would be included in the final model, as well as the "review" / "deliverable_validation" activities, which are now considered as matched since they contain the same set of matched activities. Before adding "do_first_version" too, a second inconsistency should be dealt with: keeping or not keeping the "document_production" as an extra level of abstraction. Let's assume here that it is not kept. A recursive call would then be made on the "review" activity (or "deliverable_validation"), where a new type of inconsistency would be found: Bob's view contains a sub-activity ("client_review"), which does not appear in Peter's view. The elicitor would then have the choice of whether or not to keep this activity. Figure 3 shows the resulting merged model based on the decisions taken above.

The list of inconsistencies presented to the elicitor is built by merging all inconsistencies detected when analysing each pair of views. The detection of inconsistencies is performed by evaluating a set of boolean characteristics on each child of the node being visited, in each view, with respect to every other view. These characteristics define the different types of inconsistencies. Table 3 shows the different inconsistencies handled, how they are detected, and the possible resolutions to the inconsistencies as presented to the elicitor.

This set of criteria was developed using an approach similar to the one used for the component matching algorithm (i.e., through a thorough analysis of real-world examples). It should be noted that during the validation of our work (see Chapter 7), we did not find additional cases not already covered here. Also, when creating the set of inconsistency cases, we ensured that no other cases could be discovered in theory, by ensuring that all possible combinations of values for C1 to C8 were addressed: each possible combination is mapped to one and only one case, and the impossible combinations are rejected (e.g., it is impossible to have the combination ¬C1 ^ C5, because if M does not exist, it cannot be a leaf).

After the entity decomposition for each type of entity is resolved, we can then do the other kinds of inconsistency resolution (i.e., the ones related to entity names, relationships, and attributes, introduced earlier). For these other types of inconsistency, the approach taken is similar to entity decomposition resolution. The difference is in the complexity of the basic types. In the case of the entity names, there are only two cases: the names are either the same or they are different. Similarly, we have only two cases for inconsistencies related to relationships: the relationship is either in all the views or not in some of the views. The same apply for inconsistencies related to attributes.

For inconsistencies related to names and attributes, important information to take into account in this resolution is the number of modifications made to the initial view during the entity decomposition resolution (number of subentities or subtrees added/deleted). For example, the "review" activity in Bob's view (see Figure 1) is exactly the same as in the final model (see Figure 3): it contains two levels of review, one with the IT team and one with the client. In Peter's view however, the similar entity ("deliverable_validation") has a different meaning in the sense that it contains the "modify_deliverable" activity as well, but not the "client_review". This difference makes the name "review" probably a better choice to be used in the final model. This information (about the number of modifications with the original view) is displayed in the window presenting the inconsistency to the elicitor.

Table 3. Types of inconsistencies related to "entity decomposition"

This table shows the types of inconsistencies that can occur between the decompositions of the given two views, indicates the criteria for detecting these inconsistencies, and suggests possible resolutions.

Inconsistency types (and cases with no inconsistencies)	**Criteria to detect it (see legend below for details)**	**Possible solutions**
Case 1: missing element	$\neg C1 \wedge \neg C2 \wedge C3$	Element added or not
Case 2: detail missing	$(C1 \wedge C4 \wedge \neg C5 \wedge \neg C7)$ $\vee (C1 \wedge \neg C2 \wedge \neg C4 \wedge C5)$	Decomposition (details) added or not
Case 3: finer decomposition	$\neg C1 \wedge C2 \wedge C6$	Extra level of decomposition added or not
Case 4: different grouping	$\neg C1 \wedge C2 \wedge \neg C6$	Take any of the grouping proposed by one view, or give a new one
Case 5: different decomposition	$C1 \wedge C2 \wedge C7 \wedge \neg C8$	For each element not always under the same parent, choose its parent
Case 6: all details taken from outside (leaf)	$(C1 \wedge C2 \wedge C5)$ $\vee (C1 \wedge C4 \wedge C7)$	(see case 5)
Case 7: all details taken from outside (non-leaf)	$(C1 \wedge C2 \wedge \neg C5 \wedge \neg C7)$ $\vee (C1 \wedge \neg C2 \wedge \neg C4 \wedge C7)$	(see case 5)
Case 8: different details	$\neg C1 \wedge \neg C2 \wedge \neg C3$	Take any of the decomposition proposed by one view, or give a new one
Case 9: no inconsistency (leaf)	$C1 \wedge C4 \wedge C5$	N.A.
Case 10: no inconsistency (non-leaf)	$C1 \wedge \neg C4 \wedge \neg C5 \wedge C8$	N.A.

Legend:

Assuming that:
X is the element currently under investigation
M is the element matched to X in the second view (if it exists)
L is the depth of X in the tree decomposition
S(E) is the set of all elements matched, that are under the subtree rooted at the element E given (including E if it is matched)

The criteria are:
C1 – X is matched (i.e., M exists)
C2 – X has at least one descendant that is matched
C3 – X has at least one sibling or descendant of a sibling that is matched
C4 – X is a leaf
C5 – M is a leaf
C6 – S(X) can be composed exactly by taking the union of one or more S(Ai), where Ai in any element of the second view at depth L
C7 – M has at least one descendant that is matched
C8 – S(X) = S(M)

As these inconsistencies are resolved, the information (proper name, attribute values, and relationships) is added to the merged model at the same time.

5 Discussion

A couple of key issues arise from the algorithms described above: (i) the suitability of the modeling notation used, and (ii) the scalability of the overall approach. These are discussed in the following subsections.

5.1 Modeling Notation

The algorithms have been implemented in a general way using entity-relationship diagrams (ERD), augmented with mandatory type specification for entities and relationships. This is only the *underlying* data structure. Models in other notations (e.g., dataflow, STD, UML, etc.) could be mapped into ERD, thus facilitating wider applicability of the underlying system (V-elicit). For example, a class diagram in the UML notation could be easily mapped to the ERD notation, using the following entity types: "class", "attribute", "method", etc[15]. Relationships of the following types could be used: "class contains attribute", "class is-associated-with class", "class inherits-from class", etc. In general, any graphical notation containing nodes, arcs, and attributes, can be stored in the ERD structure. The versatility of the ERD notation was the primary reason for its selection as our core data structure.

One drawback with the ERD notation is its limited set of "advanced" modeling concepts. For example, the ERD does not have a special notation for entity decomposition or entity generalisation. In such cases, special keywords for types have to be used ("is-composed-of" and "is-a" respectively here), and the merging algorithms have to deal with such types as special cases. In the description of the algorithms above, the entity decomposition aspect has clearly been handled through leaf and non-leaf comparisons when computing activity scores (matching algorithm in Section 3.2), and through a specific set of inconsistency types related to entity decomposition (inconsistency management algorithm in Section 4).

For the case of generalisation relationships, we suspect that an approach similar to the entity decomposition case would have to be used in the merging algorithms. However, we cannot be certain just yet because thus far we have not come across such a situation in our studies in the area of software processes. During our case studies for validating our methods and tool, the people we interviewed were rarely using generalization concepts for describing their tasks. The only case we found in practice was the use of the word "document" to refer to any kind of document produced (e.g., requirement document is a document, design document is a document, etc.). This is insufficient to devise a proper algorithm for such cases and validate it, and thus it is left as future work.

Other advanced modeling concepts (e.g., tertiary or n-ary relationships) suffered from the same problem: we did not find occurrences of them in practice (experience or validation work), and thus we could not justify their inclusion in the algorithms presented.

---

[15] Note that, here, we refer to a class diagram being mapped to ERD, but in the case that nodes (such as activities) represent specific instances instead of general types of entities, an object diagram used in this case could be mapped to ERD just as easily as a class diagram.

One major advantage of using such a flexible data structure (ERD augmented with mandatory user-definable types) is that it is possible to adapt the notation in a way that could help improve the algorithms (when possible). This can be achieved, for example, by specifying a better set of types to be used in the views. For instance, by replacing the type "activity" with the finer-grained types (say, "production activity", "management activity", and "quality assurance activity") the matching algorithm can be performed more efficiently by never comparing activities that are of completely different types. However, when defining such fine-grained types, it is important to ensure that their definition is clear and non-ambiguous (i.e., different elicitors would not categorize a given entity into different types), as this could result in erroneous results from the matching algorithm.

Our assumption is that the elicitors[16] are trained appropriately in the notation defined for modelling views, to increase the chances of having the views modelled in a consistent way. Also, it is assumed that when choosing the types of entities and relationships to be used in a particular elicitation effort, these are well defined prior to their use (both the syntax and the semantics), to facilitate their consistent use across elicitors. Some of the errors could be caught when checking views for consistency and completeness prior to the launch of the merging algorithms (through the checking of user-defined rules for intra-view analysis [33, 35]), but their power is limited for semantics analysis. Such feature is out of scope of this paper.

It should be noted that our algorithms could be implemented directly using another underlying data structure (e.g., UML-based), as long as such data structure meets the requirements specified above (especially the versatility in type definition, and adaptability to other notations if needed). The choice of the underlying data structure, however, does not invalidate the concepts proposed and demonstrated.

5.2 Scalability

Our approach, as presented, assumes that each person is interviewed independently in order to build one related view, and that all views are merged together at the end. Variations on this approach would have to be taken however, when the size of the process being modelled is large. One approach could be, for example, to decompose the process into relatively independent phases, and apply the view-based approach independently to each phase. Another approach would be to first elicit the model at a high level of abstraction (using the view-based approach), and then use iterations to add more details to it (by adding detailed views, and integrating them using the general view-based approach described above).

One could argue that an iterative approach, if also used with smaller views, could simplify the task of matching components and merging views. However, this is generally not a practical approach. People with appropriate knowledge are not always available, let alone repeatedly, due to other priorities. Thus the elicitation approach should generally avoid the use of numerous meetings with the same person, as this is not time efficient for that person.

[16] Note that an obvious assumption here is that an elicitor or a team of elicitors would be responsible for the view modelling, not others (e.g., practitioners or managers) who are not usually trained for this task.

It should be noted that our algorithms have been developed directly with the scalability issue in mind. To this end, we have attempted to automate the elicitation tasks as much as possible, keeping the elicitor's involvement to the minimum possible (when decisions require contextual / semantics information that the tool cannot have). Additional ideas for automating some of the decision-making tasks (currently performed manually) have been investigated, but they were considered too risky and error-prone, potentially leading to lower quality models or to the need for extensive reviews of the decisions taken.

Let us now shift our attention to related work and validation.

6 Related Work and Comparisons

In the software process area, only few elicitation approaches deal with the merging of multiple views. This type of work started in the early 1990's, with completely manual approaches. In Rombach's approach [29], for example, different views are first modelled independently, and then a cycle of reviews and modifications of the views is performed until the conflicts are resolved. However, no tool support existed for the identification of the matches and conflicts, or to help resolve conflicts and merge views into a single model. This could pose scalability problems when dealing with large process models.

In the same spirit of the use of a manual approach, Ahonen et.al. [1] relied on group sessions for capturing process information on wall-charts from appropriate stakeholders. The sessions typically lasted 3 to 5 hours, with possibility for additional sessions for major components identified requiring additional details (possibly involving a different set of experts). An advantage of group sessions is that it offers the possibility of identifying diverse opinions during the same session; however, there is also the disadvantage that people with stronger personality may affect inaccurately the outcome of an inconsistency resolution [10].

These manual approaches were soon followed by attempts to automate elicitation from multiple stakeholders [36, 38]. For example, Verlage [38] developed a similarity analysis function to help detect common components across the views, specified in the MVP-L language. The similarity analysis function does use information other than names (such as attribute values or names of objects involved in a relationship with the entity analysed – emphasizing terminology similarity though), and different weights might be used in combining scores depending on whether the views are supposed to have a significant overlap or not. The views are then compared two-by-two, for identifying inconsistencies (using a tentative set of rules). However, the user has to manually resolve these inconsistencies. Also, inconsistencies related to the abstraction hierarchies (i.e., aggregations) are not dealt with. Such hierarchies are instead kept separate, so that individuals referring back to the final process model could retain their own view of how activities are organized (or aggregated).

To our knowledge, our work, described in [36, 33], was the forerunner in the automation of view-based elicitation approaches. It involved seven key steps to the elicitation of process models: planning, view elicitation, intra-view consistency checking, identifying commonality across the views, view merging, model modifying, and quality checking. The "V elicit" tool, that supports these steps, is described

in [33]. One of its key advantages is its emphasis on a user-definable modeling notation[17], making all related features and algorithms versatile. First, the tool allow the specification of "constraints" (in first-order logic) to define what constitutes an internal inconsistency or an external validity issue. These are checked on the views prior to merging as well as on the final merged model. Second, the component matching algorithm uses all possible pieces of information (including any additional type the user may have defined) in order to calculate the similarity between entities across views. Finally, the inconsistency identification feature not only highlights problems related to name similarity and missing relationships (as in [38]), but also handles a comprehensive set of decomposition-based types of inconsistencies. From our experience, such decomposition-based inconsistencies are the most difficult ones to handle, because they cannot be solved simply by a yes/no decision (i.e., whether to keep something or not), and because various causes of the differences in activity organization require different approaches in their resolution. Overall, all of the V-elicit features were designed so that the amount of effort spent by the elicitor is reduced as much as possible.

View merging is also important in requirements engineering, where diverse views of requirements models need to be analysed. We focus on the approaches that provide tool support for either detecting overlap across views or for identifying inconsistencies.

In Lamsweerde's work [24], goals and requirements (derived from the goals) are formally defined in a temporal logic notation, and then analysed for detecting "divergences" (i.e., when there exists a boundary condition that makes a set of assertions conflicting if conjoined to it). Techniques such as backward chaining, and some heuristics and patterns, are used for such detection, which can be performed both within views and across views. They also propose ways to resolve the divergences found, such as avoiding boundary conditions, conflict anticipation, or goal weakening (mainly adding assertions to the definitions). These are implemented in the KAOS tool. Although Lamsweerde recognizes the existence of terminology clashes and structure clashes, the proposed approach does not attempt to deal with such types of clashes.

In the approach by Finkelstein, et al. [14] and Easterbrook and Nuseibeh [12], requirements from different views are first elicited in the ViewPoints framework (typically in a graphical notation) and translated into a formal (temporal logic) notation. Each view also contains a set of rules (specified in a logic notation) expressing what an inconsistency can be, and the action to be taken in the case one is found (e.g., tool invocation, or just a request for some manual work). The inconsistencies are not necessarily resolved upon discovery (instead, a history for each of them is maintained): they may be used as an indication that additional information should be elicited. It should be noted that not resolving the inconsistencies found is not an option in our case, where there is a need for a merged model free of inconsistencies. Also, it should be noted that the problem of component matching and terminology differences is not handled at all in their approaches.

[17] The V-elicit system allows the definition of the types to be used for entities and relationships, as well as the attributes the entities are allowed to contain. The "is-composed-of" and "is-a" relationship types are predefined though, in order to be treated appropriately in the different algorithms.

In [14, 12], the work focused solely on inconsistency detection, and not on matching components. At the other extreme, Leite and Freeman [25] have worked on an algorithm for matching rules. In their approach, views (specified in a rule-based language) are analysed for finding discrepancies, which are then discussed with participants for integration of views. A heuristic is presented for finding matching rules: similarity scores for facts are derived by comparing each word of the facts, and a combined score for rules is then computed using weights on facts. Our work on similarity score calculations (see Section 3) was influenced by this work.

Various other works in the area of requirements engineering focused on both aspects of view merging, namely the identification of matches and the identification and resolution of inconsistencies (not all automated though). First, Spanoudakis and Finkelstein [32] proposed the "reconciliation" method for finding overlaps between object-oriented specifications from different stakeholders, and for guiding them in the modification of the separate specifications to remove inconsistencies. Their approach is to compute the distance (similarity) between each pair of elements in the specifications and then let the user modify the specifications to reduce the distance between the similar components. Actions (manual checks) are proposed upon the analysis of the difference between the view mapping proposed by the tool and the mapping proposed by the user. In order to create a proposed mapping, the similarity score ("distance") is calculated between pairs of entities, taking into account various information structured as per the related meta-model: (a) equality of element names (or identifiers); (b) differences in classes and superclasses from which the element is an instance of (giving a higher weight for instances of classes which are higher in the abstraction hierarchy); and (c) the minimum overall distance of the entities' attributes, taken over all possible mappings of attributes. Those distances are combined into an overall distance using a given quadratic formula. It should be noted that both the distance calculation and the help provided for modifying views does not deal with decomposition relationships, which are critical in the software process area.

Barragans-Martinez et al. [4] have developed the MultiSpec methodology for merging views of software requirements expressed in state-transition diagrams. They have formally defined four merging operators: IImax (result is the union of the views – this is the type of merge required for our problem), IImin (result is the intersection of the views), IImaj (result is the set of elements found in the majority of the views), and IImaj+inc (same as IImaj but overspecified – or conflict – elements are flagged, through the use of multi-valued logic). In all cases, the first step is to identify the similar states based on the graph characteristics and on the related transitions (the transitions being mapped by name similarity only, and the states having IDs only, no name to be matched). The merging process itself checks for the presence of some graph elements but not abstraction differences across views. The inconsistencies identified can be assessed for their extent or impact. This work has been extended to prioritized views in [5].

Sabetzadeh and Easterbrook [30] present a framework for merging views expressed as general graphs in any kind of notation (e.g., i*, ERD, and state machines). In this framework, cycles of view merging and view evolution are performed, until all problems are handled. A trace is kept from each element in the merged model to the view(s) where it comes from, to help in analyzing solutions to the inconsistencies. The merging algorithm starts with the manual detection of the common elements

across the views, and the definition of interconnection diagrams that contain the mappings between the common elements. The merging process itself (or the calculation of the "colimit" of the interconnection diagram) is automated, by first building the "disjoint union" as the largest possible merged set (i.e., union of all elements of each view, as if they were all different), and then reducing this set by grouping elements that are mapped to each other in the interconnection diagram. Elements in each view are annotated with a "knowledge order" that represents the degree to which the stakeholder providing this view is knowledgeable about the given element, and such information is merged as well. The merged model created still contains the inconsistencies already present across the views, and should thus be analysed (manually) to identify such inconsistencies and resolve them.

Finally, the work by Sabetzadeh and Easterbrook on merging general graphs has been applied to Statecharts specifically in [27] (not covering parallel states though), with additional techniques provided for automatically identifying common elements across views. The approach used in their paper for matching model elements has many aspects common to ours: (a) they use a matching algorithm that computes, in two passes, similarity scores between pairs of entities, using various scores combined using weights; and (b) the matches are identified by highlighting the similarity scores above a given threshold, and then an expert is required to examine these matches. However, our approach contains a number of different features that make it more suitable for the process modeling area: (a) our name similarity function uses the fact that names (especially for activities and artefacts) are generally composed of multiple words (see Section 3.1); (b) their approach tries to match states only (with potential name differences), and assumes that transitions will always have similar names, whereas our approach is applicable to any type of model entity, which are matched one type at a time (refer to the introduction of Section 3); (c) they assume that the depth of a state is an important attribute when matching it to a state in another view, an assumption that does not hold in the process modeling area because some stakeholders or information sources see their world as a (almost) flat set of activities while others tend to use multiple levels of decomposition when organizing their activities; (d) their second pass score is computed iteratively until the similarity scores converge, while we propose not to use more than two such iterations because the use of more iterations in the views we merged lead to misleading results in the identification of proper matches (see Section 3.3); and (e) our approach uses a larger set of related information (e.g., artifacts produced or consumed by an activity, and roles associated with the performance of each activity) when computing similarity scores; such information is just not available in the statechart notation.

The area of ontology merging has also been examined for similarities with our work, although the "views" here (or different ontologies to be merged) would normally be described at a higher level than the view descriptions we have described for process models. We focus here on only the two references found which provide tool support at least as equivalent as in our method.

First, the PROMPT system [28] uses a framework similar to ours, with a matching phase involving similarity analysis followed by an inconsistency resolution phase in which inconsistencies are identified and solutions are proposed. The main difference is that in their approach the inconsistency resolution may start as soon as a few matches are identified; whereas, in our approach all matches are identified prior to the

inconsistency identification and resolution phase. The identification of matches in PROMPT though is based only on linguistic similarity of the overall element description (i.e., the identification of common terms in the concept's name and description, including synonyms), and does not take into account pre-defined types of relationships (e.g., activities producing artifacts in the process modeling area, where activities and artifacts would be considered as concepts in an ontology). Also, besides the name conflict detection performed in PROMPT, the conflicts detected are related to generalization relationships only, which do not appear much in a software process description (from our experience). Object composition (for example the activity decomposition aspect in the process modeling domain), the aspect considered one of the most difficult one to handle in process model merging, is not dealt with in PROMPT.

In [15], a fully automated system (OM) is described for merging a set of ontologies. This merging process contains also the same two phases as ours: a matching phase followed by a merging one where inconsistencies are identified and dealt with. Matches are identified mostly through linguistic similarity (as in PROMPT) but generalization relationships are also considered when finding matches. The authors of [15] admit that the matching algorithm could be improved by taking into account more types of relationships such as composition (as in our system described above). For the inconsistency resolution part, since the approach is fully automated (so no human intervention is possible to solve conflicts), an ordering of ontologies is assumed – from fundamental ones to specialized ones – and ontologies are added one by one to the most fundamental one in this order. Then, when trying to add a new concept to the merged ontology, the concept is just discarded in the case where it would lead to an inconsistency. The only inconsistencies truly dealt with are differences in abstraction levels. This assumption of being able to order views does not hold in the software process domain because amongst the set of elicited views being merge into a coherent whole model, in general, there are no priorities in terms of "levels of trust" the modeller can put in them -- they are in essence treated equally -- and that any view could contain parts (elements, relationships, semantics, etc.) that are not agreed upon to by the majority of the views. In fact, from our experience, the use of one view as the central one and adding the others to it is the source of most of the completeness problem found in a manual approach for view merging [35].

In the area of version control management, there may be, say, two developers modifying the same design diagram (e.g., UML class diagram) at the same time, which then have to be combined prior to being checked in the version control repository. These parallel modifications of the same original model may be conflicting if they were applied on the same graph elements, and so such conflicts should be detected and resolved. A number of approaches (with tool support) have been proposed for such a problem (see, for example, [2, 3, 6, 8]). These typically match common elements by names or IDs, and use the "difference models" (set of operations performed to transform the original model into one of the modified models) to find conflicts and construct the merged model (once conflicts are resolved). Unfortunately, none of these assumptions (i.e., matched elements having common names, and the existence of a common ancestor model) are valid in the area of process models because views are normally developed independently of each other, with possible mismatches in the vocabulary used. This implies that such an approach is not applicable in the merging of process models. A similar approach has also been used in the maintenance of business

process models (i.e., updating generic models to conform to the new reality, as progressively adapted in individual processes) [23], in which a set of specific executable models derived from a common high-level model are merged by reconstructing the change log for each of the executable models.

One exception (in the area of version control management) is the work by Kelter et.al. [21], in which they propose a way to match elements of different UML models without assuming common names or common IDs for matched elements (graph merging is not handled though). Their approach assumes that the elements are organized into an ordered hierarchy, such as classes containing operations, which in turn contain parameters (the ordering of the parameters being taken into account). A bottom-up approach is then used for comparing elements using similarity scores, redoing the work in the subtree when a match is found. The similarity score calculation resembles ours (see Section 3.1), in the sense that a weighted average of lower-level scores (associated with the links to related elements, for example the ratio of matched parameters for an operation) is used. However, they are not using entity decomposition information in this process. In particular, they are assuming that no element under the subtree of one matched element can be found under a different subtree in the other diagram, which is not a valid assumption in our case. Also, they indicate that links to elements not considered for matches yet (such as transitions between states in a statechart diagram) could be handled simply by reapplying the score calculations iteratively until the matches are found. However, from our experience with views from our domain (see the second variant of our algorithm described in Section 3.3), such an approach tends to produce poor results when many iterations are used. For the purpose indicated above though, many iterations would be necessary, thus limiting the use of relationships among entities of the same type in the matching process. So, the approach in [21] does not use as much related information on the elements being compared as in our method.

We also examined the work in the area of the merging of database schemas; however, the components being merged in this case (i.e., the fields within each table, and possibly the integrity constraints) were considered too different to the process model entities and relationships to be described and compared against.

Finally, in the area of model engineering, Kolovos et al. [22] propose a generic model merging operation consisting of the same merging phases as ours [33, 36]. However, the key difference with our work is that, whereas we define algorithms for finding a match or a conflict (see Sections 3 and 4), Kolovos et al. give a rule-based language (called EML) and tool support with which users are able to specify how to detect matches and how to transform the matched elements into a merged conflict-free model.

7 Validation

As described earlier, the algorithms in Sections 3 and 4 have been implemented in a system called V-elicit [33, 35]. The system was validated through two empirical studies, one a controlled study to compare our approach against existing ones, and the other to test its efficacy with industrial-scale processes.

7.1 Comparison Against Other Tools

The goal of the first study was to objectively compare V-elicit to third party modelling tools. We were particularly interested in comparing the quality of the resultant process models using these systems. This study is published in detail in [37], but we summarise it here for ease of access.

The conducted controlled study involved six graduate students, familiar with the area of software process technology. Three used V-elicit; whereas three others each used a different commercial tool. The tools had various modelling notations, from dataflow and control flow to state transition diagrams. The subjects all elicited and modelled the same three different processes in which they had adequate background. They were proficient in the use of the tool assigned to them, through training prior to the study. The processes were from unbiased sources: the ISPW6 example [19], and two from a priori and independently modelled third-party industry projects at McGill University. These processes were presented to the subjects as three different views, described in plain English.

The first quality factor investigated was "model completeness", using the following metric:

$$\text{Completeness} = (1 - (EM/TE)) * (1 - (RM/TR))$$

where: EM = number of entities missing; TE = total number of entities to be modelled; RM = number of relationships missing; and TR = total number of relationships to be modelled within the scope (entities) actually modelled (i.e., relationships with entities missing were not considered in that ratio).

Table 4. Controlled study results - completeness

Tool used (each student)	Completeness for each of the three processes			Mean completeness over the 3 processes
V-elicit	0.925	1.000	0.918	0.948
V-elicit	0.860	0.857	0.838	0.852
V-elicit	0.698	1.000	0.869	0.856
Tool 1	0.613	0.612	0.808	0.678
Tool 2	0.618	0.844	0.667	0.709
Tool 3	0.562	0.854	0.752	0.723

The results are shown in Table 4. Using a 2-way ANOVA test [17] for factors "subject" (column "tool used" in Table 4) and "process", we conclude that the results are statistically significant, with a p-value of 0.025. Additional tests on means (using the Student-Newman-Keuls range test [17]) revealed that the models produced using V-elicit were significantly more complete than the ones produced using other tools (using non-view-based elicitation approach). However, no significant difference was observed among the subjects using V-elicit, or among subjects using the other tools. The significance level used for this test was 0.05.

Other quality factors were investigated such as model consistency and accuracy, but the large differences in the completeness of the models meant that our analysis could only be qualitative. We noticed that the more incomplete models resultant from

the use of third party tools generally had much poorer consistency and accuracy compared to those developed using V-elicit.

Since the conclusion of this study, we have improved the underlying algorithms in V-elicit, but this has not changed the general view-based approach used, and the kind of interactions between the elicitor and the tool. Also, to our knowledge, none of the currently available commercial tools use a dramatically different elicitation approach (and none of them uses a view-based approach). We are thus confident that the conclusions of this controlled study still holds, implying that a view-based approach still results in higher quality process models.

It should be noted that a similar study performed more recently by Easterbrook [13] (comparing view-based vs traditional approaches for merging views, in the context of requirements engineering however) led to a similar set of findings than ours: (a) a view-based approach helps detecting all inconsistency problems; (b) a view-based approach improves the understanding of the problem domain; (c) the model built tend to be closer to what was said in the interviews (especially in the wording used); and (d) using such an approach is significantly more time-consuming however.

7.2 Industrial-Scale Elicitation

The V-elicit tool (with its improved algorithm) has also been validated through the elicitation of three industrial-scale process models from various domains, with various view sizes, and with various amount of overlap between the views.

The first one of these elicitation efforts involved a preliminary analysis process (a part of requirement engineering). The information source included interview data from three different agents: an analyst, a manager, and a client representative. The entire process contained approximately: 90 activities, 36 artefacts, and 8 roles. Each view contained only a subset of the entire process, where 14% of the activities and 17% of the artefacts overlapped across the views. The roles modelled were similar in all the views however.

V-elicit was successfully used to create a coherent, merged, model from the three described views. The algorithm for identifying common components across the views did necessitate elicitor involvement in some cases, and the use of similarity scores computed (e.g., see Table 1) reduced the search for the best matches from a large number of entities across the three views to only few possible solutions per match. Manually, this would have been an arduous and error-prone task. The inconsistency identification algorithm identified properly all the inconsistencies: 74 of them related to missing entities in some views (non-overlapping information), and 13 of them related to structural differences across the views (with no new type of inconsistency found – only the ones already identified in Table 3). In addition, the name and attribute differences, as well as missing relationships, were also identified properly. The resultant model was validated by three researchers intimately involved with the modelled process[18].

Similarly, V-elicit was used successfully in two more elicitation efforts, the first one involving a review process, and the second one involving a web interface development process. In both cases, three views were used, ranging from 24 to 48 activities

[18] The large size of the final model does not permit it to be included in the paper.

per view, and with various percentage of commonality across views (from 4% to 54%). For the first case, the data source was again interview transcripts from a previous project. For the second one, we gathered data through interviews. On average, 78% of the expected matches were found (88%, 68%, and 78% respectively for each of the elicitation efforts), and 22% of reported matches were incorrect (25%, 18%, and 23% respectively for each of the elicitation efforts). The remaining matches were easily found through the analysis of few of the similarity scores. It should be noted that in all three case studies, no "tuning" of our approach or algorithm were required.

After these studies had been conducted, we used the views gathered to assess whether improved results of the matching algorithm (see Section 3) could have been achieved using different weights in the calculation of the similarity scores. In essence, for each weight, we recomputed the similarity scores (see Sections 3.1 and 3.2) using different values for that weight (while keeping the other weights constant), and recorded the number of correct matches found. Table 5 shows the range of optimum values (i.e., resulting in the highest numbers of correct matches found) for each weight, obtained from the assessment of the views resultant from the three studies. It should be noted that the weights used during the studies, chosen intuitively, did fall in these ranges. In a similar way, we also determined that the best cut-off value at the end of the First-pass score is 1.5, and that the best cut-off value for the selection of matches (at the end of the matching algorithm) is 0.5. It should be noted however that these are only suggestions from the three industrial case studies, and that confirmatory investigations are a subject of future studies.

Table 5. Suggested weight ranges

Pass	Weight range
Name	1-3
Roles	0-2
Artefacts	0-2
Activities	0-1
Attributes	0-2

8 Conclusion

In the realm of modelling large, complex process models, seeking views of the object processes from different sources are one way to elicit the full model in a piecemeal manner. However, views add a new level of complexity in terms of overlaps and inconsistency across the views, which need to be addressed, preferably, with the aid of automated tools to make the tasks more manageable.

This paper describes "view-merging" algorithms, which form a core part of a view-based model elicitation system [36, 35, 33]. In our investigations, the models were those of software processes, captured as entity-relationships diagrams augmented with type information. The described algorithms aid in identifying overlaps and inconsistencies across a set of views and in presenting possible resolutions. By interacting with the model elicitor (using the V-elicit system [33] that implements these algorithms), the described algorithms help in incrementally building a unified, coherent,

process model. We expect that any tool embodying the described algorithms would typically be used in a larger context for process elicitation, which would include such tasks as human-oriented negotiations and conflict resolution requiring human input. These human-oriented issues are not discussed in this paper though examples can be found in [33].

While there were two parallel efforts to ours [38, 29] in the early to mid-90s, when we actually implemented the embodying V-elicit system [36], more recently, there has been such interest in the area of requirements engineering (RE) [4, 30], where the concept of view-based requirements has long been embraced by the RE community though it lacked system implementation. Furthermore, approaches have been proposed in other areas such as ontology merging [28, 15] and version control management [2, 3, 6, 8, 21]. However, these are not suitable for merging process models. Reasons include certain assumptions and restrictions assumed in these areas, that are not valid when dealing with: (i) diverse process views (with potentially inconsistent use of vocabulary), and (ii) views built individually rather than from a common ancestor model. In this respect, our effort is the most comprehensive to date both in terms of technical details in the algorithms (see Sections 3 and 4) for merging process models as well as system implementation (see [33]). Section 6 describes related work and specific comparisons. For future work, one might consider enhancing view-based modelling into a new dimension, that of linking specific (instance) models to more generalised models in a variety of application domains.

References

1. Ahonen, J.J., Forsell, M., Taskinen, S.-K.: A Modest but Practical Software Process Modeling Technique for Software Process Improvement. Journal of Software Process Improvement and Practice 7(1), 33–44 (2002)
2. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
3. Altmanninger, K.: Models in Conflict - Towards a Semantically Enhanced Version Control System for Models. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 293–304. Springer, Heidelberg (2008)
4. Barragáns-Martínez, A.B., Arias, J.J.P., Vilas, A.F., Duque, J.G., Nores, M.L., Redondo, R.P.D., Blanco-Fernández, Y.: Composing Multi-perspective Software Requirements Specifications. Int. J. Software Engineering and Knowledge Engineering 18(1), 119–153 (2008)
5. Barragáns-Martínez, A.B., Arias, J.J.P., Vilas, A.F., Duque, J.G., Nores, M.L., Redondo, R.P.D., Blanco-Fernández, Y.: Composing Requirements Specifications from Multiple Prioritized Sources. J. Requirements Engineering 13, 187–206 (2008)
6. Brosch, P., Langer, P., Seidl, M., Wimmer, M.: Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder. In: ICSE Workshop on Comparison and Versioning of Software Models, pp. 55–60. IEEE CS Press, Los Alamitos (2009)
7. Bush, M.W.: Process Assessments in NASA. In: 13th Int. Conf. on Software Engineering, pp. 299–304. IEEE CS Press, Los Alamitos (1991)
8. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)

9. Curtis, B., Kellner, M.I., Over, J.: Process Modeling. Communications of the ACM 35(9), 75–90 (1992)
10. Dalkey, N., Helmer, O.: An Experimental Application of the Delphi Method to the Use of Experts. Management Science 9(3), 458–467 (1963)
11. Deiters, W., Gruhn, V.: Software Process Technology Transfer - A Case Study Based on FUNSOFT Nets and MELMAC. In: 8th Int. Software Process Workshop, pp. 50–52. IEEE CS Press, Los Alamitos (1993)
12. Easterbrook, S., Nuseibeh, B.: Using Viewpoints for Inconsistency Management. Software Engineering Journal 11(1), 31–43 (1996)
13. Easterbrook, S., Yu, E., Aranda, J., Fan, Y., Horkoff, J., Leica, M., Qadir, R.A.: Do viewpoints lead to better conceptual models? An exploratory case study. In: 13th IEEE International Conference on Requirements Engineering, pp. 199–208. IEEE CS Press, Los Alamitos (2005)
14. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency Handling in Multi-Perspective Specifications. IEEE Trans. on Software Engineering 20(8), 569–578 (1994)
15. Guzman-Arenas, A., Cuevas, A.-D.: Knowledge accumulation through automatic merging of ontologies. Expert Systems with Applications 37(3), 1991–2005 (2010)
16. Hardgrave, B.C., Armstrong, D.J.: Software Process Improvement: It's a Journey, Not a Destination. Communications of the ACM 48(11), 93–96 (2005)
17. Hicks, C.R.: Fundamental concepts in the design of experiments, 4th edn. Saunders College Publishing, Philadelphia (1993)
18. Humphrey, W.S.: Managing the Software Process. Addison-Wesley, Reading (1989)
19. Kellner, M.I., Feiler, P.H., Finkelstein, A., Katayama, T., Osterweil, L.J., Penedo, M.H., Rombach, H.D.: ISPW-6 Software Process Example. In: First Int. Conf. on the Software Process, pp. 176–186. IEEE CS Press, Los Alamitos (1991)
20. Kellner, M.I., Hansen, G.A.: Software Process Modeling: A Case Study. In: 22nd Annual Hawaii Int. Conf. on System Sciences. Software Track, vol. II, pp. 175–188. IEEE CS Press, Los Alamitos (1989)
21. Kelter, U., Wehren, J., Niere, J.: A Generic Difference Algorithm for UML Models. In: Liggesmeyer, P., Pohl, K., Goedicke, M. (eds.) Software Engineering 2005. LNI, vol. 64, pp. 105–116. GI (2005)
22. Kolovos, D.S., Paige, R., Polack, F.: Merging Models with the Epsilon Merging Language (EML). In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)
23. Küster, J.M., Gerth, C., Förster, A., Engels, G.: Detecting and Resolving Process Model Differences in the Absence of a Change Log. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 244–260. Springer, Heidelberg (2008)
24. van Lamsweerde, A., Darimont, R., Letier, E.: Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Trans. on Software Engineering 24(11), 908–926 (1998)
25. Leite, J.C., Freeman, P.A.: Requirements Validation Through Viewpoint Resolution. IEEE Trans. on Software Engineering 17(12), 1253–1269 (1991)
26. Madhavji, N.H., Holtje, D., Hong, W., Bruckhaus, T.: Elicit: A Method for Eliciting Process Models. In: 3rd Int. Conf. on the Software Process, pp. 111–122. IEEE CS Press, Los Alamitos (1994)
27. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Statecharts Specifications. In: 29th International Conference on Software Engineering, pp. 54–64. IEEE CS Press, Los Alamitos (2007)

28. Noy, N.F., Musen, M.A.: PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In: 17th National Conf. on Artificial Intelligence, Austin, TX, pp. 450–455. American Association for Artificial Intelligence (2000)
29. Rombach, H.D.: Practical use of formal process models: first experiences. In: 8th Int. Software Process Workshop, pp. 132–134. IEEE CS Press, Los Alamitos (1993)
30. Sabetzadeh, M., Easterbrook, S.: View Merging in the Presence of Incompleteness and Inconsistency. J. Requirements Engineering 11, 174–193 (2006)
31. Spanoudakis, G., Constantopoulos, P.: Elaborating Analogies from Conceptual Models. Int. Journal of Intelligent Systems 11(11), 917–974 (1996)
32. Spanoudakis, G., Finkelstein, A.: Reconciling requirements: a method for managing interference, inconsistency and conflict. In: Annals of Software Engineering, vol. 3, pp. 433–457. Springer, Heidelberg (1997)
33. Tassé, J., Madhavji, N.H.: View-Based Process Elicitation: a User's Perspective. Journal of Software Process Improvement and Practice 6(3), 125–139 (2001)
34. Tullos, R.E.: Assessment of Similarity Indices for Undesirable Properties and a new Tripartite Similarity Index Based on Cost Functions. In: Palm, M.E., Chapela, I.H. (eds.) Mycology in Sustainable Development: Expanding Concepts, Vanishing Borders. Parkway Publishers (1997)
35. Turgeon, J.: A View-Based System for Eliciting Software Process Models. Ph.D. Thesis, McGill University (1999)
36. Turgeon, J., Madhavji, N.H.: A Systematic, View-Based Approach to Eliciting Process Models. In: Montangero, C. (ed.) EWSPT 1996. LNCS, vol. 1149, pp. 276–282. Springer, Heidelberg (1996)
37. Turgeon, J., Madhavji, N.H.: View-based vs Traditional Modeling Approaches: Which is Better? In: Conradi, R. (ed.) EWSPT 2000. LNCS, vol. 1780, pp. 131–137. Springer, Heidelberg (2000)
38. Verlage, M.: An Approach for Capturing Large Software Development Processes by Integration of Views Modeled Independently. In: 10th Int. Conf. On Software Engineering and Knowledge Engineering. Knowledge Systems Institute, Skokie (1998)

Model Checking Programmable Router Configurations

Luca Zanolin[1], Cecilia Mascolo[2], and Wolfgang Emmerich[3]

[1] Google UK Ltd
76 Buckingham Palace Road
London SW1W 9TQ, UK
lucazanolin@gmail.com
[2] Computer Laboratory
University of Cambridge
15 JJ Thomson Avenue
Cambridge CB3 0FD, UK
cecilia.mascolo@cl.cam.ac.uk
[3] Dept. of Computer Science
University College London
Gower Street
London WC1E 6BT, UK
we@acm.org

Abstract. Programmable networks offer the ability to customize router behaviour at run time, thus increasing flexibility of network administration. Programmable network routers are configured using domain-specific languages. In this paper, we describe our approach to defining the syntax and semantics of such a domain-specific language. The ability to evolve router programs dynamically creates potential for misconfigurations. By exploiting domain-specific abstractions, we are able to translate router configurations into Promela and validate them using the Spin model checker, thus providing reasoning support for our domain-specific language. To evaluate our approach we use our configuration language to express the IETF's Differentiated Services specification and show that industrial-sized DiffServ router configurations can be validated using Spin on a standard PC.

1 Introduction

Most routers in use in the current Internet are rather simple; they receive packets from one network interface, investigate the packet header that encodes the target IP address and forward them to a router that is closer to the target. Most current routers implement the so-called best-effort model, which implies that all packets are *equal citizens*; they are all processed in the same manner. There are, however, applications, such as audio-conferencing or video-on-demand, that could be improved by quality of service (QoS) guarantees provided by the network. Moreover, there are different classes of users; companies might be prepared

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 473–491, 2010.

to pay a premium for performance and bandwidth guarantees. These guarantees cannot be given with the current best-effort model of the Internet.

To address this question, a relatively novel strand of network research investigates programmable networks [2], which give up the assumption that every network packet is handled in the same way by a router. Instead, the router executes a program that controls more intelligently how network packets are to be handled and forwarded to other networks. Such programmable routers can then be used to implement QoS Internet standards, such as the Differentiated Services model (DiffServ) [1]. DiffServ suggests marking packets in order to identify their service class at boundary routers so that then network traffic can be shaped by delaying low class packets or even dropping them. The identification of service classes, conditions on shaping and dropping, and traffic management can then be implemented in router programs and thus different qualities of network services can be provided by a set of programmable routers.

The rise of programmable routers implies a number of interesting software engineering research questions. Firstly, it is desirable to provide a high-level and application specific programming language to allow network administrators to write router programs at appropriate levels of abstractions. At the same time, this language must be efficiently executed in order to avoid compromising network and routing performance. Secondly, router programs need to be changed on a regular basis in order to introduce new services without having to shut down the routers. Thirdly, the network typically contains a large number of routers that might be quite heterogeneous and we would like to hide this heterogeneity from network administrators. Finally, prior to updating a single router or a set of routers with a new router program, network administrators may want to check that their programs do not compromise network performance and reliability.

The main contribution of this paper is a solution to the last question. We describe a high-level router programming language that can be used to define the packet processing performed by a programmable router. We define the semantics of the language in an operational manner by mapping it to Promela, the specification language defined for Spin [8]. We then show how linear temporal logic (LTL) [13] can be used to specify safety and liveness properties for a router and demonstrate that Spin can efficiently and effectively check whether a given router program meets these properties. We discuss an evaluation of our approach using a set of DiffServ router programs and show how the model checking support is integrated into a network administration environment. By focusing on the software engineering aspect of this inter-disciplinary project, this paper presents an interesting case study for the systematic engineering of an application-specific configuration language whose definition has been inspired by graphical architecture description languages, the definition of its operational semantics and the provision of reasoning support using model checking techniques.

The paper is structured as follows. In Section 2, we briefly introduce the notion of programmable routers and describe our router configuration language. We define the operational semantics of the configuration language in Section 3 by mapping it to Promela. In Section 4 we show how we use that mapping to

prove router properties with Spin. In Section 5, we present the architecture and the implementation of our tool and we evaluate it in Section 6. Section 7 compares our approach to related work and in Section 8 we discuss future research directions of this project.

2 Programmable Routers

A programmable router can be configured in order to exploit the network resources according to different requirements. There are many applications of programmable routers, including firewalls that can rapidly respond to denial of service attacks, virtual private networks, traffic shaping, and provision of configurable quality of services.

We have implemented a programmable router, called `Promile` [12]. `Promile` is composed of two layers: the Router Kernel and the XML-bAsed Middleware (XAM). The kernel deals with packet forwarding, while XAM manages the router configuration. XAM presents the administrators with an abstraction of the routing machine, allowing them to modify the router to provide new services or modify old ones. We call the router abstraction *configuration* because it configures how the router manages packets. The configuration is a high level abstraction, described by a set of *modules*.

When a packet is received by the router, it is given to a module, that, after applying a function, sends it to the next module. When the packet reaches the last module, the packet is re-injected into the network and forwarded to the next hop. According to its class and its header, a packet can go through different modules allowing the router to provide different services to different packet flows.

In order to implement qualities of service at the network level, the IETF has recommended the Differentiated Services (DiffServ) model [1]. In this paper, we use DiffServ to illustrate our approach to model checking programmable routers. DiffServ assigns a *class* to each packet and DiffServ routers handle and forward packets according to the class of the packet. The class of the packets is not used as a priority, but it identifies the type of service required for handling the packet.

Figure 1 is a simple example of a DiffServ router configuration. In this scenario, we suppose that the router is able to provide only two kinds of services: *video conference* and *file download*. Briefly, the first service has to guarantee the packet delivery with a given fixed bandwidth (the service is not interested into infinitely increasing throughput, as this would not improve the quality of a video conferencing application). The second service is interested in maximizing the packet transmission. As a consequence, we can delay packets from a video conference only if they are above the bandwidth requirement, while if the network is congested, we can delay or even drop the packets from a download session. Using a DiffServ architecture, each service is mapped into a different class, which is recorded in the IP packet header.

As shown in Figure 1, the *Receiver* module receives packets from the network and forwards them to the *Marker* module. According to the fields stored inside

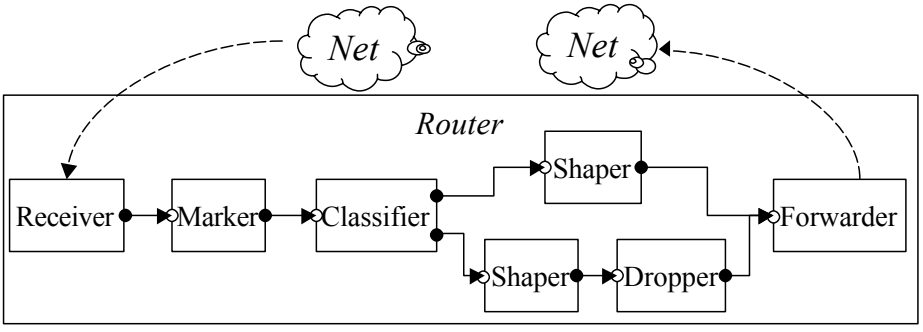


Fig. 1. Router Configuration

the packet header, the *Marker* assigns packets to a particular class. In this example, we can argue that *Marker* assigns the class according to the source host; we assume that *Marker* knows which hosts are downloading data and which are in a video conference session; obviously, this is a big simplification, but showing how to set up a DiffServ network is beyond the scope of this paper. More details are provided in [12].

When packets have been dealt with by *Marker*, they are sent to the *Classifier* module that, according to their class, routes them into the *upper* path, that is, the video conference/first class path, or into the *lower* path, that is, the download/second class path. When the packets are routed into the upper path, they go through the *Shaper* module, that is able to delay them. Otherwise, the packets can be routed into the lower path and go through the *Shaper* and *Dropper* modules where the packets can be delayed (*Shaper*) or even dropped (*Dropper*) in order to reduce network congestion. We now use this example to briefly introduce the concept of our application-specific language.

2.1 Modules

Modules are the basic building blocks of router configurations. By interconnecting modules and parameterizing their behaviour, we define the configuration of the router that provides the desired services.

A module (Figure 2) is a black box that applies a function to incoming packets. Modules communicate with other modules through *gates*. A packet is received from an input gate and sent through an output gate. A module can have multiple input gates in order to distinguish different kinds of packets. Most modules have both input and output gates, with two exceptions. A module with only output gates receives incoming packets from the network; this kind of module is called *source module* and usually describes an input network interface. Similarly, a module that only has input gates forwards packets into the network and is called *sink module* and it models an output network interface. When a packet reaches

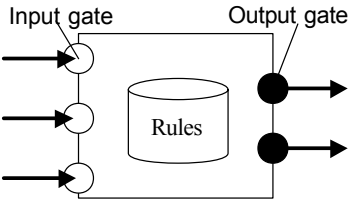


Fig. 2. A `Promile` Module

the router, it is firstly managed by a source module and, after being processed by a configuration, is sent to the next router or its destination by a sink module.

2.2 Rules

A module can be configured through a set of rules. According to these rules, the module applies its internal functionalities only to a sub-set of the incoming packets and it routes them through its output gates. A rule identifies packets through a set of fields related to the packet header and to the environment. The field sets may be different depending on the module's behaviour; this flexibility allows to precisely identify when a module functionality should be applied.

Table 1. Rules

Num	Source		Destination		Class
	TCP	IP	TCP	IP	
1.	10	128.16.8.57	20	128.16.10.27	1
2.	10	128.16.10.27	*	*	2
3.	1000	*	1000	*	10
4.	1200	*	*	*	10

Table 1 shows a sample rule set of the *Marker* module. It describes how classes (last column) are assigned to packets based on source and destination IP addresses. The first rule identifies the packet class by specifying both source and destination packet address and assigns it to the *first class*. The second, third and fourth rules specify only a sub-set of the fields. The third rule, for instance, specifies that packets that are sent from TCP port 1000 to any destination on TCP port 1000 will be marked with class 10. Using these rules, we can flexibly parameterise the behaviour of individual module instances.

2.3 Connections

Connections start from an output gate and lead into an input gate describing the packet flow among the modules. Connections can describe complex module

graphs that should comply with the following properties. Firstly, any output gate must be connected to an input gate to fully specify the packet forwarding from module to module. Secondly, only one connection can start from an output gate. This ensures that the packet forwarding is deterministic. Finally, multiple connections may end in the same input gate and they may define elaborated router configuration with loops.

2.4 Configuration Update

The module instances, their interconnections, as well as the rules that have been used to parameterise the behaviour of module instances determine the *configuration* of a router. The management of these configurations for a `Promile` router is an important aim of the work described in this paper. The network administrator is able to change the router configuration without stopping it and, moreover, without any traffic disruption; as a consequence, the network can quickly react to changes in its environment, updating its configuration according to the application requirements.

When a network administrator, or an automatic tool, wants to update the router configuration, they have to deal only with the router abstraction provided by `Promile`. Prior to defining a new configuration, the administrator wants to be sure that the new configuration is meaningful. In order to do so, we have to support the definition of invariant properties.

2.5 Properties

The properties that network administrators might want to prove can be divided into three groups: *routing*, *service*, and *performance* properties.

Routing properties are concerned with the router and they define how the packet should be managed independently from provided services (i.e. *video conference support*); these properties guarantee that the router always works correctly and it is eventually able to handle new incoming packets. As these properties are not service-specific, they are almost the same for all the routers in the network and they are usually defined by the router developers. For instance, a routing property could be concerned with the module graph; as we have described, the module connections may be cyclic and this might cause packets to loop infinitely in the graph if this behaviour is not prevented by the configuration rules of modules contained in the cycle.

Service properties are concerned with the services that are to be implemented with a router configuration. With these properties, we want to guarantee that the configuration provides the services correctly. As these properties are service dependent, they can change from router configuration to router configuration. In general, they can be defined by administrators. For instance, an administrator can require that some functions are never applied to particular packets (i.e, never drop packets of premium users).

Performance properties are concerned with the router performance and they try to maximize router throughput through an optimisation of the configuration. Both service and routing properties are necessary, while performance properties are not vital. However, as optimal performance is an important feature for any router, we argue that a router has also to satisfy this kind of properties. Performance properties may be defined by the network administrator, but also by the router developer. A performance property, for example, may want to forbid that a packet is first marked and then dropped by the router, as this would make the router wasting time marking the packet that, instead, should have been dropped straight away.

Another performance property that should be checked is concerned with rules and modules. According to the router packet flow, some rules could be irrelevant, as no packets match them. In order to increase the router speed these should be removed. We should also check if all the modules are reachable by at most one packet, otherwise we should remove them.

3 Operational Semantics

In order to prove that the router satisfies desired properties, we translate the router configuration into a formal specification. We can then apply model checking techniques to validate the router configuration against the desired properties. By defining this translation, we give an operational semantics to the router configuration language.

We split this translation into two steps. The first step translates the router configuration into an *intermediate representation*, which describes each module through a set of *components*; the second step produces a Promela specification that defines the operational semantics and can be model checked by Spin. We assume that the `Promile` module types may be developed by third parties. In order to enable model checking of router configurations that instantiate such third party module types, module developers have to define the semantics of their module types. The first advantage of our two stage mapping is that we have defined the intermediate representation in such a way that it is relatively straightforward for a module developer to define the semantics of a module type using that intermediate representation.

We now describe the intermediate representation and its mapping to the Promela language.

3.1 Intermediate Representation

The intermediate representation supports four component types: *source* (SO), *selector* (SE), *executor* (EX), and *sink* (SI) (Figure 3). A *source* (Figure 3a) forwards new packets to the next component; this component does not actually create packets, but it directly takes them from the network. A *selector* (Figure 3b) receives packets and routes them to one of possibly several components

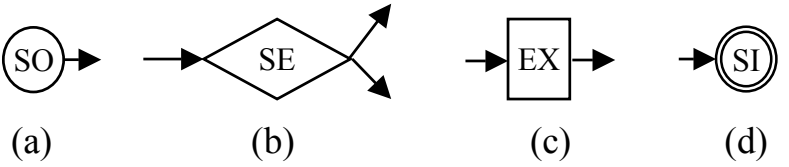


Fig. 3. Components

according to selection rules that it applies. The selector is working as a *multiplexer*, where the rules describe how the incoming packet must be forwarded to the next component. A selector has one input and a list of numbered outputs; if a received packet matches no rules, it is sent through the default output (0). An *executor* (Figure 3c) receives packets and forwards them after applying a function. An executor applies the same function to all packets that it processes. A *sink* (Figure 3d) defines the end of the packet path. Arrows denote connections between components and they determine packet flows.

Third party module developers have to provide translations between their module types and this intermediate representation in order to provide semantics for their modules. For instance, a DiffServ module of type *Dropper* could be mapped into the intermediate representation shown in Figure 4. The semantics of this module type is described using a selector component (SE) that chooses the packets that should be dropped and forwards them to the executor component (EX) that applies the module specific function (i.e., drops packets) and sends them to the sink component (SI) where the packet disappears and it is not forward any further.

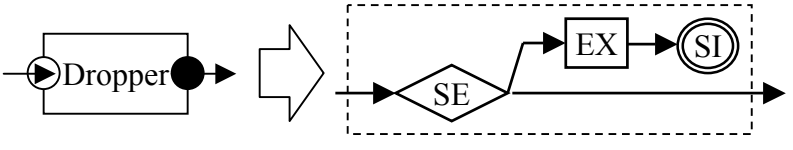


Fig. 4. Translation of a Dropper Module

Figure 5 shows how the router configuration in Figure 1 is mapped using the four component types that we have introduced. The rules defined in Table 1 need to be incorporated into the model as they define the specific behaviour of the different modules. This operation is trivially done importing them into the *selector* component of each module. The *selector* is the only component containing rules.

The next step is to translate this intermediate representation into a formal specification language (i.e., Promela) in order to be able to model check the router configuration.

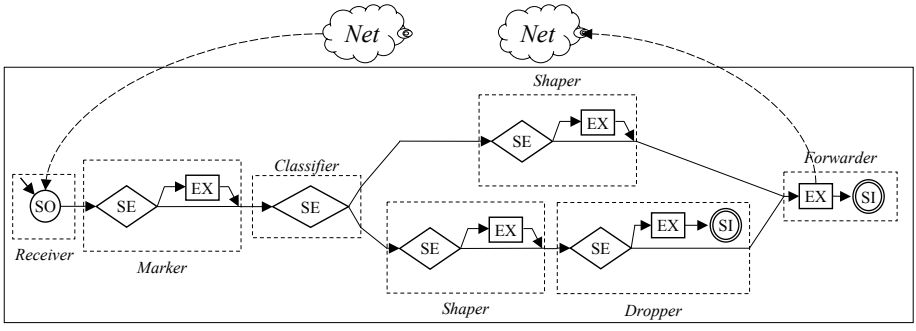


Fig. 5. Component Graph

3.2 Promela Specification

The Promela specification that is automatically derived from an intermediate representation is composed of a set of processes that can communicate with each other. Each process corresponds to a component and it formally describes how a packet is dealt with and how it is forwarded to the next process; there is a one to one relationship between the intermediate component types and Promela process types (*process source*, *process selector*, *process executor*, and *process sink*).

Promela processes communicate with each other through a set of global variables, that describe the packet properties. These global variables are accessible by all processes that can read and modify them. We have introduced some extra variables for modelling the packet forwarding and the environment; one of them is the variable `exec` that describes the functions that are applied to packets; by monitoring the value of this variable, we can check which functions are applied and in which order.

All processes have a similar structure; a process waits until it receives a packet, then executes its statement, such as routing packets to different processes, or applying functions to them (i.e., dropping). For instance, Figure 6 shows a simplified version of the Promela code for a *selector process* that corresponds to a selector component; the selector process chooses the process to handle the incoming packet; through a sequence of `if` statements, the process checks the value of the packet fields and identifies the next process (setting the variable `next_proc`). The `else` statement is executed if the packet fields do not match any of the other conditions (line 6). We assume that at most one condition is true for a particular value of the packet fields, avoiding a non-deterministic evolution of the process.

A subset of rules defined in Table 1 are mapped into this selector process. For instance, the rule on line 4 corresponds to the rule on line 3 in Table 1, where the rule describes packets from TCP source port 1000 going to TCP destination port 1000.

```
1. proctype p_2() {
2.  (next_proc==p_id_2);
3.  if
4.  ::(TCPS==1000 && TCPD==1000)->next_proc=p_id_3;
5.  ::(TCPS==1200) -> next_proc=p_id_3;
6.  ::else -> next_proc=p_id_4;
7.  fi;}
```

Fig. 6. Selector Process

The router model, which is translated into the Promela specification, handles one packet at a time. This simplification of the model does not limit the analysis of a real router, as we can assume that the presence of a packet does not influence the management of other packets inside the physical router; in fact, the concurrent management of packets speeds up only the router throughput without any influence in the packet handling.

4 Model Checking Promile

Given the Promela specification we can now prove properties on the router configuration using the Spin model checker.

4.1 Abstractions to Reduce the State Space

The router model must be checked exhaustively to guarantee that the router configuration is acceptable. This implies checking that all packets reaching the router are handled correctly.

To attain this goal, we cannot generate the set of all possible packets, with all possible header field settings, as this would lead to exponential growth of the state space; even considering only the field values referenced by the rules, the packet growth is still exponential. In fact, if we call $\mathbb{P}$ the set of the packets that we need to inject into the router model, and we consider the worst-case scenario where all the rules refer to different field values, the cardinality of this set is shown in (1), where r is the average number of the rules in each module, m the number of modules, and f the number of fields in each rule.

$$|\mathbb{P}| = (m * r)^f \tag{1}$$

$$|\mathbb{P}| <= \sum_{i=1}^{k} \binom{m}{i} * r^i + 1 \tag{2}$$

$$k = min(maxpath, f)$$

However, the cardinality of $\mathbb{P}$ can be limited considering groups of packets, which we will call *packet flows*, instead of single packets, observing the structure of the router configuration and exploiting the relationships among the rules.

Following this idea, we can generate less packet flows and reduce the cardinality of the set $\mathbb{P}$ as shown in (2). The cardinality of $\mathbb{P}$ still grows exponentially, but the base is reduced from $m * r$ to r; the exponent is also reduced and is the minimum between the number of fields and the maximum number of the router modules that a packet can traverse ($maxpath$). Moreover, the cardinality of $\mathbb{P}$ is described by an upper bound, reached only in the worst-case. We now prove that (2) is an improvement over (1), that is we prove that:

$$\sum_{i=1}^{k} \binom{m}{i} * r^i + 1 < (m * r)^f \tag{3}$$

We approximate assuming the difference between the two terms is bigger than 1:

$$\sum_{i=1}^{k} \binom{m}{i} * r^i < (m * r)^f \tag{4}$$

With:

$$\begin{aligned} m, r, f, k > 0, m \geq k, f \geq k \\ k = min\{f, maxpath\} \end{aligned} \tag{5}$$

$$\sum_{i=1}^{k} \binom{m}{i} * r^i < \sum_{i=1}^{k} \binom{m}{i} * r^f < (m)^f * r^f \tag{6}$$

$$\sum_{i=1}^{k} \binom{m}{i} < (m)^f \tag{7}$$

The first term of (7) is indeed less than m^f because:

$$\sum_{i=1}^{k} \binom{m}{i} \leq \sum_{i=1}^{f} \binom{m}{i} = m + \sum_{i=2}^{f} \binom{m}{i} \tag{8}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} \sum_{i=2}^{f} \frac{1}{i!} \tag{9}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} (\frac{1}{2} + \frac{1}{2*3} ... + \frac{1}{2*3*...*f}) \tag{10}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} (\frac{1}{2} + \frac{1}{2*2} ... + \frac{1}{2*2*...*2}) \tag{11}$$

As the last term in (11) is less than 1 we can approximate:

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} \tag{12}$$

and for f larger than 2:

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m^f \tag{13}$$

4.2 Flow Generation

In this section we show how reasoning on packet flows instead of single packets allows us to limit the exponential growth of our model. In Section 6, we will then show that flows sufficiently abstract the state space to enable model checks of real-life router configurations.

As packets are identified by a set of fields, a flow can be described by a subset of fields. Following this approach, we group together packets that have some of the fields set to the same value, ignoring the other fields that can assume *potentially* all the possible values. For instance, if the rules inside the router use only two fields (the TCP ports of source (TCPS) and destination (TCPD)), a flow can be described as in (14). Flow f_1 contains all the packets, with fields set to specific values of TCPS and TCPD. In this case, the flow is said to be *fully-defined*, as all fields have a value. On the other hand, flow f_2 is *undefined* as the value of TCPD is not set; we say that flow f_1 is *more defined* than f_2, as it is a subset of f_2 (16).

$$\mathbf{f_1} = \{(TCPS, TCPD) | TCPS = 1 \wedge TCPD = 1\} \tag{14}$$

$$\mathbf{f_2} = \{(TCPS, TCPD) | TCPS = 1 \wedge TCPD = null\} \tag{15}$$

$$\mathbf{f_1} \subset \mathbf{f_2} \tag{16}$$

In order to prove that the router behaves correctly, we check the router configuration against a set of flows that cover all different paths inside the router. We need to simulate the generation of enough flows so that all the rules inside the modules are matched by at least one flow. By injecting all these flows into the router, the model checker can validate the configuration and pinpoint any router undesired behaviours.

Flows are generated using a Promela specification derived from the intermediate representation of the router. Each component of the intermediate representation is translated into Promela process and, in order to distinguish these processes from the ones that describe the router configuration, we call processes that describe flow generation *network processes*, while *router processes* those that describe the physical router. The connections among the network processes are the same as in the router processes.

The source component is translated into a network process that generates a fully-undefined flow that represents the packets that can reach the router. This flow will be defined more accurately while traversing the selector network processes; the selector network processes correspond to selector components and redefine the packet flow according to their rules. We say that a flow matches a rule when all fields have a compatible value; a flow field is compatible with a rule field if it has the same value or if it is undefined. When a packet flow matches a rule, the selector network process redefines the setting of fields to the value described in the rule. As a flow can match more than one rule, the selector network process randomly chooses one of them.

A simplified network process of a selector component is shown in Figure 7. At line 4 the rule has two fields (TCP source and destination port of the packet); in order to match this rule, the flow has to have the fields undefined or set to same values of the rule.

```
1.proctype np_2() {
2. (next_proc==np_id_2);
3. if
4. ::((TCPS==1000||TCPS==null)
     &&(TCPD==1000||TCPD==null))->
          TCPS=1000;TCPD=1000;next_proc=np_id_3;
5. ::(TCPS==1200||TCPS==null)->
          TCPS=1200;next_proc=np_id_3;
6. ::else -> next_proc=np_id_4;
7. fi;}
```

Fig. 7. Selector Network Process

The sink component is mapped into a network process that behaves as a bridge between the network and the router processes; when it receives a flow, it simply forwards it to the router processes. An executor component is mapped into a forwarding process that receives a flow and forwards it to the next process; the functions applied by this component are irrelevant for the flow generation and they are not modelled.

The number of different flows determines the state space of model checker. The formula in (2) describes the upper bound of the flow set cardinality for the following reason. Consider a simple scenario where packets have only one relevant field (A), and where each module has the same number of rules r. The network source process defines a flow where the field A is set to *null* (i.e., a fully-undefined flow). This flow then traverses the other modules and arrives at the network sink process. We can argue that a *fully-defined* flow (i.e., where all fields have values) can evolve only through deterministic steps, as there is at most one rule that can match it; on the contrary, a flow where the field is undefined can match all the r rules and then can evolve in r different ways; moreover, when a flow matches a rule it becomes fully-defined, as the field is being set by the rule (and there is only one field per packet in the example). As a consequence, if a

non-defined flow reaches every process that has r rules, we obtain (17), where m is the number of modules in our model.

$$\mathbb{P} <= r * m \tag{17}$$

If we want to fully validate the router configuration, we need to also generate a flow matching no rules, so in reality we need to generate $r * m + 1$ flows. This corresponds to the formula in (2) with $f = 1$.

The scenario with $f = 1$ can be extended for a generic value of f. A flow that has f fields can match f different rules in the worst-case, i.e., each rule sets the value of only one field of the flow. For instance, let us consider a flow that is described by two fields, A and B, and a router configuration composed of two modules that have r rules each; moreover, we also assume that the rules inside the first module refer to field A, while the ones in the second module to field B. We now inject a fully-undefined flow; as field A has no value, the flow can match any rule and its field may be set to the value described in one of the rules. The evolution of the flow is non-deterministic and can evolve in $r + 1$ different ways (matching r different rules or matching none). When this set of $r + 1$ flows reaches the second module, the flow evolution is more complex. Each flow can match any of the rules, as field B is undefined. Moreover, the flow can traverse the module without matching any rule. Therefore, the $r + 1$ flows can match all the rules, and we obtain $(r + 1) * r$ flows, or they do not match any rule and we are left with $r + 1$ flows. Then we can deduce the formula:

$$(r+1) * r + r + 1 = r^2 + 2 * r + 1 = \binom{2}{2} * r^2 + \binom{2}{1} * r + 1$$

This formula is formula (2) with $f = 2$. The idea can be extended considering m modules and f fields obtaining formula (2) .

4.3 Proving Router Properties

We can prove that the router configuration complies with a set of defined properties. The property validation is based on a simplification of the router model; the router model is able to manage only one packet flow at a time, and moreover, after that, it is turned off. Obviously this behaviour is not the same of a physical router, but, if we can prove that the router model is able to handle correctly a single packet flow, then the real router will be also able to handle a sequence of packets.

The prove of some properties is embedded into the specification; for instance, we can deduce which rules are not used by reasoning about the unreachable code of the Promela specification, and find out which rules or modules should be removed from the router configuration. Moreover, through the *assertion* mechanism supported by Spin and embedded in the specification, we can verify that no packet is going to loop inside the router simply asserting that no module will receives two or more packet flows.

Finally, we can define generic properties using Linear Temporal Logic (LTL). Through LTL, we can define the sequences of functions that have to be applied to every packet and forbidden sequences. In this way, we can formalize the *router*, *service*, and *performance properties*, as defined in Section 2.5. For instance, a *service* property is described in (18). This property requires that the packet that belongs to a particular host will never be dropped; the packets are identified through the term $fromHost(A)$. The other LTL property (19) is concerned with router performance. This property forbids dropping a packet that was previously marked inside the same router.

$$\Box(fromHost(A) \implies \Box(\neg dropPacket())) \tag{18}$$

$$\Box(markPacket() \implies \Box(\neg dropPacket())) \tag{19}$$

5 Implementation

In Section 2 we have introduced the architecture of `Promile`, that is composed of two layers: an Xml-bAsed Middleware (XAM) and the Router Kernel. The Router Kernel manages the packet forwarding process, while XAM is the middleware that manages updates to the configuration of the router. The requirements of these two layers are different as they have different goals. XAM is portable and flexible to run on different hardware platforms, while the kernel maximizes router throughput. XAM is designed to work on different Programmable Routers. XAM interfaces the network administrator with the router kernel. The model checker is a separate component and is supposed to run on a different processor, for performance reasons. The administrator can use a visual tool, shown in Figure 8, to set up a new configuration that is then model checked prior to being sent to the kernel. These steps are completely transparent to the network administrator, who only deals with the graphical router configuration language and uses a wizard to define service-specific router properties.

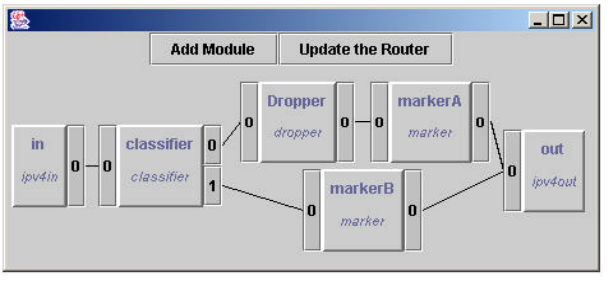


Fig. 8. Administrator Visual Tool

XAM stores the router configuration as an XML document that can be manipulated using open source libraries that implement the WWW consortium's Document Object Model (DOM) and XPath. XAM has also to prove that the

new configuration complies with a set of defined properties and, in order to achieve that, it has to transform the configuration into the intermediated representation and then into the Promela specification that can be analysed by the model checker. The router configuration is stored as an XML document and through a style sheet transformation (XSLT), XAM translates the configuration into the intermediate representation. We use style sheets to enable third party module designers to provide a semantics mapping of their modules to the intermediate representation (Figure 9).

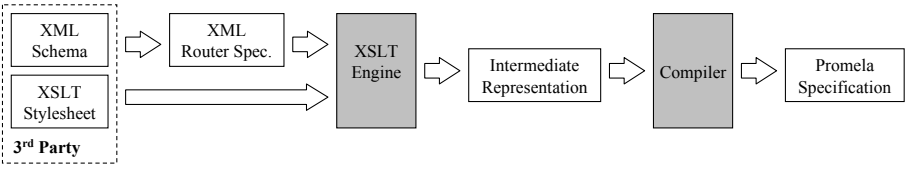


Fig. 9. Translation Steps

The second translation step into Promela is implemented in Java for performance reasons. If Spin finds an erroneous configuration, the error is shown to the administrator via the visual tool. A full description of the architecture of XAM and the Router Kernel can be found in [12].

6 Evaluation

Our work aims at minimizing the validation time of a router configuration. This operation must be flexible enough to accommodate any network change and must be relatively fast to be used interactively with the graphical administration tool. According to the requirements of the network administrator, the validation can be done on-line before applying the operation, or off-line if the router update cannot be delayed (e.g. during a *denial of service attack*).

We have tested several realistic router configurations and in particular we have used two different configurations: a 1-level tree and a sequence of modules. We have chosen these two configurations so that our approach reaches the best and the worse performance, respectively. In fact, the number of generated packet flows is related to the number of modules that a packet can traverse; in the first case, there are only 2, the root and the leaf, while in the second case all the modules. The two configurations have the same number of modules (seven), and the rules concern the same fields (three), while the number of rules goes from 7 to 8,500. We have chosen to change the number of rules, as, from our experience, the number of modules or fields is usually three or four orders of magnitude lower than the number of rules; it is then more relevant to prove how our approach scales according to rules, instead of other parameters. As these two configurations produce the worse and the best performance of our tool, they also represent an upper and a lower performance bound for a router configuration with the same number of modules, rules, and fields.

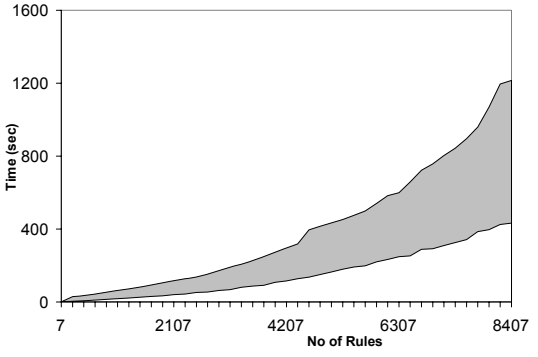


Fig. 10. Model Checker Performance

In Figure 10, the lower and the upper performance bounds are enclosed in the gray area. Interpolating the result, we have obtained the two equations shown in (20) and in (21). Both of them have a small coefficient of the second order and in particular (20) can be approximated to a line.

$$y = 0.2609 * x^2 - 0.3319 * x + 8.4093 \tag{20}$$

$$y = 0.7653 * x^2 - 4,6192 * x + 56,287 \tag{21}$$

These tests have been performed on a PC equipped with a Xeon@1.7GHz processor and with 1 GB of RAM. Note that a configuration with seven modules and a total of 8,500 rules is a rather complex one for this specific application domain. We consider it remarkable that on a relatively small off-the-shelf PC we are able to validate the worst case configuration in less than 20 minutes.

7 Related Work

The related work to this project belong to two different research areas: networking and software engineering. From the networking research area, there are two interesting projects: Click [9] and Router Plugins [5]. Click and Router Plugins are implementations of programmable routers, where the packet paths inside the router (i.e, the modules and their connections) can be configured; the abstraction provided by Click is more complex than our kernel, as there are more port types in order to describe the behaviour of the module more precisely. Furthermore, Click and Router Plugins do not provide a middleware layer to support the network administrator during the configuration process. However, we believe that the principles outlined in this paper could also be applied to these projects. From the software engineering perspective, model checking techniques, as data and model abstraction, are been investigated in [3]. The Bandera toolset [4] can verify that generic Java code complies with a set of properties defined by users; this is achieved through model slicing and abstraction; in particular, the abstraction is manually driven by users through a graphic interface. In our approach,

the abstraction of the model is provided by the middleware, that shows to the model checker tool a graph of modules; as the domain of our tool is always the same, we have implemented an automatic abstraction of the input data (i.e., the packets) that is completely transparent for the network administrator and provides better performance than a general purpose one; however this approach is domain specific and cannot be applied to generic scenarios.

Model checking techniques are also used in networking protocol research [7]. In this domain, the system is distributed and it is relevant to cope with its embedded concurrency; through model checking as shown in [6], the network protocol is specified using Promela and validated through Spin. The advantages of this approach are mainly two. Firstly, the network protocol is designed using a specification language (i.e. Promela). Secondly, the specification can be validated using a model checker (i.e., Spin). Spin is used to check the correctness of the network protocol, but it does not cope with the network configuration. In fact, it assumes that the network topology is correct and that it should work correctly also if there are some errors. Spin checks the robustness of the protocol, but it does not check the network environment. The aim of our work is complementary: we do not manage the router life-cycle, but we model check its configuration in order to guarantee that it will work correctly and that it will comply with the required properties.

Our work has been influenced by graphical ADLs, and in particular by Darwin [10], which describes a system configuration through components, ports, and connectors. Like Darwin, we are able to dynamically update a configuration and apply model checking techniques to it. The difference is that our configuration language has been explicitly geared toward its application domain and is directly executed by our programmable router's forwarding engine. Also, Darwin uses reachability analysis of labelled transition systems for model checking [11], while we use Spin, a more powerful model checker that allows us to check any LTL formula.

8 Conclusions and Future Work

In this paper we have described an approach to management of programmable networks that takes advantage of model checking techniques in order to prove that router configurations are consistent and safe. The model checker is integrated into a visual tool that allows the network administrator to manage a network, confidently update configurations at run time. The tool is based on a formal description that can be translated into Promela, the specification language of Spin. Exploiting the model checker Spin, we can prove that the router configuration is correct or, report errors to the network administrator through a visual tool. In terms of analysis, we are currently able to prove that a router is providing the right services in the right way, but we cannot check that the whole network is working as required.

The work presented in this paper shows how application-specific languages can be designed by applying software engineering principles, how their semantics

can be defined and most importantly that it becomes possible to apply model checking to real-life problems by exploiting domain-specific abstractions.

The next steps of this work will go in two different directions; through the visual tool, the network administrator will be able to *draw* the router properties instead of writing LTL formulas; moreover, the extension of the tool domain is relevant in order to move the verification from a single router to a network of routers.

Acknowledgments. We would like to thank Karen Page for her assistance in the proof. We thank the `Promile` group and Licia Capra for her comments on an earlier draft. We acknowledge the financial support of EPSRC, BT, and Kodak.

References

1. Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., Weiss, W.: IETF-RFC 2475 - An Architecture for Differentiated Services (December 1998)
2. Campbell, A.T., Meer, H.G.D., Kounavis, M.E., Miki, K., Vicente, J.B., Villela, D.: A Survey of Programmable Networks. ACM SIGCOMM Computer Communications Review 29(2), 7–23 (1999)
3. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Transactions on Programming Languages and Systems 16(5), 1512–1542 (1994)
4. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu Robby, C.S., Zheng, H.: Bandera: extracting finite-state models from java source code. In: The 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, pp. 439–448 (June 2000)
5. Decasper, D., Dittia, Z., Parulkar, G., Plattner, B.: Route plugins: A software architecture for next-generation routers. IEEE/ACM Transactions on Networking 8(1), 2–15 (2000)
6. Gauthier, E., Boudec, J.-Y.L., Oechslin, P.: SMART: A many-to-many multicast protocol for ATM. IEEE Journal of Selected Areas in Communications 15(3), 458–472 (1997)
7. Holzman, P.: Design and Validation of Network Protocols. Prentice Hall, Englewood Cliffs (1991)
8. Holzmann, G.: The SPIN Model Checker. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
9. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F.: The Click modular router. ACM Transactions on Computer Systems 18(3), 263–297 (2000)
10. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995)
11. Magee, J., Kramer, J.: Concurrency: Models and Programs – From Finite State Models to Java Programs. John Wiley, Chichester (1999)
12. Meer, H.D., Emmerich, W., Mascolo, C., Pezzi, N., Rio, M., Zanolin, L.: Middleware and Management Support for Programmable QoS-Network Architectures. In: Marshall, I.W., Nettles, S.M., Wakamiya, N. (eds.) IWAN 2001. LNCS, vol. 2207. Springer, Heidelberg (2001)
13. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. Foundations of Computer Science(FOCS 1977), Providence, Rhode Island, pp. 46–57 (October 1977)

Architectural Issues of Adaptive Pervasive Systems

Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi

Politecnico di Milano
Deep-SE Group - Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 - 20133 Milano, Italy
{caporuscio,funaro,ghezzi}@elet.polimi.it

Abstract. Pervasive systems are often made out of distributed software components that run on different computational units (appliances, sensing and actuating devices, computers). Such components are often developed, maintained, and even operated by different parties. Applications are increasingly built by dynamically discovering and composing such components in a situation-aware manner. By this we mean that applications follow some strategies to self-organize themselves to adapt their behavior depending on the changing situation in which they operate, for example the physical environment. They may also evolve autonomously in response to changing requirements. Software architectures are considered a well-suited abstraction to achieve situational adaptation. In this paper, we review some existing architectural approaches to self-adaptation and propose a high-level meta-model for architectures that supports dynamic adaptation. The meta-model is then instantiated in a specific ambient computing case study, which is used to illustrate its applicability.

Keywords: Pervasive Systems, Software Architecture, Software Evolution, Context-aware Adaptation.

1 Introduction

Many modern advanced applications are developed as pervasive systems that support ubiquitous, continuous and smart interactions among humans, autonomous devices, and the environment, to realize what is often called *ambient intelligence*. Such systems, sometimes also called *open-world systems* [1], are characterized by a highly dynamic software architecture: both the components that are part of the architecture and their interconnections may change dynamically, while applications are running. New components may in fact be created by component providers and made available dynamically. Components may then be discovered, deployed, and composed at run time, removing pre-existing bindings to other components. Applications are often highly distributed, i.e., components are deployed and run on different computational units that may not just be traditional computers, but also appliances, sensing and actuating devices of different kinds. In many cases, the components that constitute an application are

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 492–511, 2010.

also operated and run by decentralized and autonomous entities. It has become common to use the terms *services* for such components and *service-oriented architecture (SOA)* to indicate the architectural style. In the SOA case, applications do not have a single ownership and coordination point. Services can only be invoked remotely through their interface. They are otherwise seen from other parts of an application only as black-boxes [13].

Dynamic architectures of the kind we described above are created to support the adaptive and evolutionary situation-aware behaviors that characterize pervasive systems. Sometimes it may be useful to distinguish between *adaptation* and *evolution*. Adaptation refers to the actions taken at run time and affecting the architectural level, to react to the changing environment in which these systems operate. In fact, changes in the physical context may often require the software architecture to also change. As an example, a certain service used by the application may become unaccessible as a new physical environment is entered during execution and a new service may instead become visible. Or a certain service may be changed unexpectedly by the owner of the service and the change may be incompatible with its use from other parts of the application. Evolution instead refers to changes that are the consequence of requirements changes. For example, a 3-D interface becomes available and must be used instead of a previously used traditional interface. In the rest of this paper, most of our examples refer to adaptation, although our approach can also work for certain kinds of evolution. In general, long-lived pervasive systems require that applications should follow some strategies to

- detect the relevant changes in the situation in which they operate, such as the physical environment (or even the changing requirements), and
- react by self-organizing themselves and adapting their behavior in response to such changes.

Adaptive pervasive systems raise many challenges to software engineering. They stress the known methods, techniques, and best practices to their extreme and introduce new difficult problems for which new solutions are needed. The notions of variability and adaptation must permeate all phases, from requirements to design and validation, and even run time. Indeed, the clear and clean traditional separation between *development time* and *run time* becomes blurring. Traditionally, changes are handled off-line, during the maintenance phase. In the new setting, they must be also handled autonomously at run time, as the application is running and providing service. To achieve that, software systems must be able to reason about themselves and their state as they operate, through adequate reflective features available at run time. They must be able to monitor the environment, compare the data they gather against an expected model, and detect possible situational changes. Whenever a deviation is found, an adaptation step must be performed, which modifies the software architecture. For example, the adaptation step might simply perform a new deployment and re-bindings to different components, or re-binding to different external services. In other cases, the adaptation strategies may be more complex.

This paper focuses on software architectures that support run-time adaptation. More specifically, it illustrates a general, high-level reference meta-model and then illustrates how it can be instantiated and adapted to develop a specific case-study in the domain of assisted living.

The paper is structured as follows: Section 2 describes related work. Section 3 illustrates a practical hypothetical example of an adaptive system that serves as a case-study in a pervasive computing setting. Section 4 introduces the most important features of the proposed meta-architecture. Section 5 describes how the concepts of the meta-architecture are instantiated in the proposed solution for the case-study. Finally, Section 6 provides some conclusions and outlines future work directions.

2 Related Work

Research on dynamically adaptable and evolvable software systems became very active in recent years. In the early 2000s, IBM promoted a vision, called *autonomic computing* [10], which focuses on a new generation of software systems that can manage themselves to achieve their goals in a changing world, through *self-configuration*, *self-optimization*, *self-healing*, and *self-protection*. Although an overview of autonomic computing is out of our scope, and would clash with space reasons, we will narrow down the focus of our analysis of related work to architectural solutions enabling self-management. Research in this area has been focusing on two main issues. On the one side, there has been an exploration of the architectural *styles* that best support evolution, due to intrinsic characteristics of the style. On the other, research has focused on the *mechanisms* that can be exploited to achieve adaptation, given a specific architectural model or a specific style. According to [17], "An architectural style defines a vocabulary of components and connector types, and set of constraints on how they can be combined." By focusing on architectural styles, it is possible to focus on evolution from an abstract and high-level viewpoint which may enable systematic and even formal reasoning.

Let us first observe that in the general case, if no specific constraints are assumed on an architecture, a run-time change that requires dynamic updates of components or connectors may require suspension of (parts of) an application to achieve some desirable level of consistency. Managing suspensions can be very complex. This problem has been faced elsewhere in the literature [11,23].

There are styles, however, that facilitate dynamic adaptation. C2 [14] is a well known example of an architectural style that achieves this goal. C2 introduces a sharp distinction between computation and communication and strictly constrains how the application can be built. In particular, every communication among computational units (components) occurs via bus-like connectors, thus minimizing component interdependency. The style also imposes topological constraints: components can consume data from only one connector and produce output data only on one connector. Connectors, instead, can accommodate any number of components or other connectors. Thus every communication is carried

out in an asynchronous way through messages put on and read from connectors. The C2 style is well suited to supporting dynamic adaptation and evolution. The application can be easily modified through the addition and/or removal of components, which can be carried out without suspending any computation.

Other styles than C2 have been scrutinized by Taylor et al. [21] . In that paper, a number of architectural styles used by state-of-art software systems are evaluated according to following criteria:

- How and how much the system's behavior can be changed;
- How long the system's evolution takes to be effective;
- How the state of the system is changed when that system evolves;
- In which environment the system is executing;

Examples of examined styles and corresponding systems are: the publish-subscribe style [7], implemented for example by Siena; the REST style [8] used for web browsing; the CREST style [6] adopted by AJAX and other JavaScript-based technologies.

Regarding the mechanisms that can be superimposed to an architectural model or style to achieve dynamic adaptation, two main research lines emerged so far. The former is about all the approaches that exploit *planning* techniques to cope with unexpected situations, and the second one is based on reactive rule systems. An early example of planning-based approach has been proposed by Traverso and Pistore [22], which synthesizes plans starting from OWL-S process models and a set of prioritized goals. These plans cope with non-deterministic outputs of services by synthesizing `if-then-else` constructs to cover all the possible outputs prescribed by the OWL-S process model [12]. Another planning-based approach is described by Sykes et al. [19,20], where a plan is synthesized starting from goals and available operations. The difference lies in that the non-deterministic output of actions is handled by synthesizing a *reactive* plan. A reactive plan is a plan that for each logical state of the system from which the goal can be reached, prescribes the action to be taken to move towards the goal. In this way, even if the system should deviate from the expected behavior, as long as the goal is reachable by the current state the plan can still suggest a way to achieve the goal.

The other main approach to adaptation is rule-based, e.g., the Rainbow framework [9]. Rainbow is based on the use of a run-time architectural model. Logical probes can be deployed on the running system to gather data useful to enable system evolution and adaptation. Data coming from probes can be aggregated and then used to update the run-time model. To accomplish this task ad-hoc components called *gauges* are introduced. Invariants can be stated about the run-time model and reactive rules can be written to try to enforce them. Reactive rules directly manipulate the run-time architectural model and these changes are automatically reflected in the controlled application.

Another rule-based approach has been developed in our group. It introduces an autonomic element called SelfLet [3] , its model and its developing framework. A SelfLet is characterized by a *goal* and by a course of action (specified through a

finite state machine) to achieve it, called *behavior*. In a world populated by SelfLets, each of them tries to achieve its goal by exploiting functionalities available locally or requesting the needed functionalities to other SelfLets. Such request can be fulfilled both by a SelfLet providing the functionality as a service or by a SelfLet teaching the requestor how to solve the problem. SelfLets are also able to evolve and adapt on the basis of their internal state and the environment they work in. To do so, an *autonomic policy* is specified through reaction rules. Rules can manipulate a SelfLet's behavior by:

- changing the way a service is offered or requested;
- installing a new local service;
- modifying a behavior by adding, deleting or replacing states and transitions in the finite state machine describing the SelfLet.

Combining local *behaviors* and *reactive rules*, a software architect can design a system able to achieve a global goal relying on a SelfLet population making only local decisions. Rule-based approaches are in general computationally lighter than planning-based approaches, but offer only little support to handling unforeseen situations since no reactive rule has been written for them.

An approach, the A3 framework, that cannot be classified neither as planning-based nor as rule-based has also been developed in our group by Baresi and Guinea [2]. A3 is a component-based framework where components are organized in groups, created by application designer to decompose the global task into local sub-tasks. Each group elects a supervisor which both handles communication from and to the group and monitors the group itself. Groups are dynamically created and destroyed and no constraint is imposed on their topology. The grouping dynamism allows to cope with changed situations(e.g., a group with too many components to be effectively handled by a single supervisor). The absence of topological constraints enables the creation of overlapping groups or even hierarchies to more effectively share information among groups. Thus centralized systems' pitfalls are avoided and at the same time the grouping mechanisms make the framework scale very well even for a large number of components.

The approach we illustrate here differs from most of the related work in that it aims at providing a meta-architecture through which the proposed architectural styles and the adaptation mechanisms may be instantiated.

3 Scenario

This section introduces a scenario describing an adaptive embedded system. In particular, it addresses functional evolution of devices embedding on-demand software, namely smart set-top boxes for pay-per-view television. The scenario is further used subsequently to describe the proposed approach.

A *set-top box* is a device that, connected to a source of signal (e.g., telephone line or TV cable) and a television, decodes the signal into contents to be displayed on the television screen. Next generation set-top boxes will also connect to TCP/IP networks by enabling users to browse the web, as well as to access and consume services on-demand.

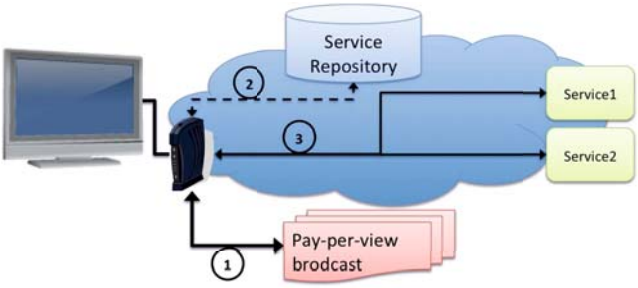


Fig. 1. Scenario

Figure 1 shows a typical scenario where the smart set-top box (1) accesses the contents broadcast by the TV provider, (2) accesses and browses a trusted *Service Repository* (also published by the TV provider) listing the available services, and (3) interacts with the selected services by connecting with the relevant *Service Providers*. When the set-top box is switched on for the first time, it must be configured by the user by selecting the required features. Hence, a basic facility provided by the set-top box supports connection with the TV provider's *Service Repository* (dashed line in Figure 1), a catalog from which the user may select the set of services he or she want to access. As we will see, the *Service Repository* behaves both as a *Registry*, which contains the full descriptions of the services, and as a *Repository* of components that may be downloaded and services provided by the TV provider that can be invoked remotely from the set-top box.

In this setting, possible use cases are: (*i*) *TV on-demand*, where the user chooses which type of contents he or she is interested in, and (*ii*) *Service on-demand*, where the user selects the set of services offered by third-party providers.

3.1 TV On-Demand: Pay-Per-View Home Cinema

The set-top box can access both free and pay-per-view TV contents. While free contents can be directly accessed, in order to consume pay-per-view contents, a user must possess the rights to access the selected channels. In particular, let the user be interested in the "Home Cinema" channel. Then, the actions performed are:

1. User accesses the *Service Repository*, browses the list of available channels and selects the "Home Cinema" pay-per-view channel.
2. Once chosen the channel and paid for it, the set-top box automatically downloads from the *Service Repository* the software component needed to decode the desired contents (e.g., *HC*), which is broadcast in an encrypted format. Indeed, this component is chosen by taking into account both the subscribed channel and the context of the set-top box (e.g., television properties, user requirements).
3. The *HC* component is then deployed into the set-top box.

In this scenario, the set-top box might be reconfigured to support both *software evolution* and *context-aware adaptation*. As an example of evolution, consider a scenario in which a component is downloaded and deployed to improve service fruition (e.g., new codec version, component bug-fix). As an example of adaptation, consider instead the ability of reconfiguring the set-top box with respect to the actual context (e.g., different type of television, server side updates).

3.2 Service On-Demand: eHealth Emergency Management System

As stated above, the set-top box can be used not only to download new contents, but also to access third-party services. For instance, an e-health service might be available to manage health alarms from its subscribers. The user may in this case use the set-top box to send a "health alarm" to the nearest hospital. Let us assume that the service of interest, namely the "eHealth Emergency Management" service (*HEM*), is available in the *Service Repository*. *HEM* is a composite service whose workflow is shown in Figure 2. Namely, it involves a number of standalone activities: (*i*) a *Hospital Yellow Pages* activity (*HYP*) that, given a geographical location, returns a list of the nearest hospitals, (*ii*) the *eHealth Management* (*HM*) provided by the hospital and, (*iii*) a *Display Result* (*DR*) that notifies the workflow results to the final user.

Fig. 2. The "eHealth Emergency Management" workflow

More precisely, the following sequence of steps is performed:

1. User accesses *Service Repository*, browses the list of available services in the entire registry and selects "eHealth Emergency Management" (*HEM*).
2. *HEM* is implemented as a composite service, where the activities mentioned above are provided by means of either local components (e.g., *DR*) or third-party remote services (e.g., *HYP* and *HM*). Hence the *Service Repository* sends to the set-top box the code that implements the workflow of Figure 2 through which both local components and external services are invoked when the user presses a certain button of the remote control.

Also in the case of service on-demand, the set-top box may reconfigure its architecture to achieve both software evolution and context-aware adaptation. As an example of evolution, consider a scenario in which an activity is updated (e.g., a new requirement is added or the activity interaction protocol is modified). On the other hand, as an example of adaptation, consider a scenario in which the set-top box is moved from a location to a different one. By changing the set-top box location (i.e., *context*) the list of the nearest hospitals returned by

HYP changes accordingly and, in turn, the alarm must be sent to a new *HM*. Since the new *HM* could rely on a different interaction protocol – e.g., by means of dial-up – the set-top box needs to be reconfigured in order to *adapt* to the new environment (further discussed in Section 5).

4 A Reference Meta-architecture

In this section we describe the fundamental concepts and properties that characterize software architectures for adaptive pervasive systems. In particular, we crystallize them as a reference high-level meta-architecture that specifies the main distinctive concepts upon which we can define architectural models of self-adaptive systems. Further, Section 5 illustrates a possible architectural solution for the scenario presented in Section 3 and will show how the architectural model conforms to the meta-architecture defined herein. Indeed, the proposed meta-architecture can be instantiated in different ways and even partially in other practical scenarios.

The meta-architecture illustrated in Figure 3 is reminiscent of, and an enrichment of, the abstract structure of an autonomic element [10]. In fact, it defines the main features that characterize a software architecture of applications that may evolve and adapt their functionalities with respect to a change of either their requirements or their surrounding environment.

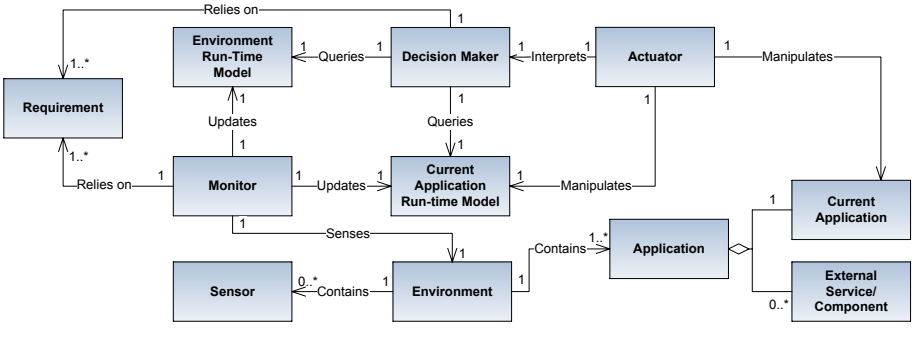


Fig. 3. Meta architecture for self-adaptive systems

The cornerstone of the proposed meta-architecture is the *Requirement* entity, which defines the initial input steering the application assembly, as well as the application run-time behavior. Indeed, it defines the set of properties that an application must satisfy at run time. It is worth to note that in this context, *evolution* refers to the ability of changing requirements at run time, whereas *adaptation* refers to the ability of satisfying the requirements in spite of changes within the execution environment. This twofold role of requirements demands for (*i*) a *Decision Maker* that assembles an abstract description of the application able to satisfy the requirements, and (*ii*) a *Monitor* that is in charge of

collecting data about the application's run-time behavior to verify whether the requirements are satisfied or not during the execution.

The *Decision Maker* is an entity that, relying on *Requirements*, is able to synthesize and assemble an abstract description of the application. The description is abstract in the sense that it does not deal with implementation details but it is a process-like description of the application's behavior specifying (*i*) which activities must be executed to accomplish the task specified by *Requirements*, (*ii*) how the activities interact with one other, and (*iii*) the logic needed to assemble the activities – e.g., referring to the service on-demand scenario in Section 3.2, the *Decision Maker* is in charge of synthesizing the workflow in Figure 2. Furthermore, since these operations must be accomplished also at run time, *Decision Maker* must consider the application's run-time situation to analyze on-the-fly if the application's behavior adheres to the requirements. Specifically, a *situation* is a composite view of both the current state of the application and its surrounding environment. *Decision Maker* retrieves such data by querying *Application Run-time Model* and *Environment Run-time Model*, respectively. If the situation does not satisfy the requirements (e.g., when the set-top box is moved to a new location, the health alarm must be sent to a new hospital), a new abstract description is synthesized and passed to the *Actuator*, which in turn is responsible for assembling and deploying the actual application (*Current Application* – CA). Specifically, *Actuator* interprets the abstract description provided by *Decision Maker* and handles all the technological means needed to build and bootstrap the new application – e.g., locating and accessing the software artifacts implementing the workflow activities in Figure 2. Such separation of concerns makes abstract descriptions technology-agnostic by effectively decoupling the general description of the application from technology-specific actuation. Hence, abstract descriptions generated by *Decision Maker* can be stored and subsequently reused every time they are required, irrespectively of the technology-specific execution environment. As an example, given an abstract description we can generate two equivalent applications, implemented by means of two different technologies, by providing such an abstract description to two different and technology-specific actuators.

Furthermore, to properly make decisions about the run-time reconfigurations, *Decision Maker* needs to query both the application and the environment run-time models. In order to be effective, such models should reflect reality as faithfully as possible. The monitor collects run-time data from the environment; specifically, data gathered from *External Services/Components* that are not part of *Current Application* and sensor data that provide relevant information about the physical environment – e.g., the set-top box position. The result of monitoring can be an update of *Environment Run-time Model* or of *Current Application Run-time Model*. In Figure 4 *Sensor* denotes an abstraction of a device that provides physical context information. Figure 4 also distinguishes between the *Current Application* (CA) and the other *External Services/Components* it interacts with, which may change over time.

5 Scenario Implementation

As introduced in Section 4, the proposed architectural-model for run-time software evolution is centered around the use of a run-time model as an abstraction of the evolving application and the surrounding environment. Further, several entities, making use of such a model, are devised in order to accomplish run-time software evolution. In this section, we apply such a reference meta-architecture to the scenario described in Section 3 and, in particular, we describe how its entities are mapped to this specific use case.

5.1 Application and Environment Run-Time Models

As mentioned earlier, to properly make decisions concerning the dynamic reconfigurations to accomplish, the system must be able to reason about itself and the state of the environment it operates in. To this extent, *Current Application Run-time Model* and *Environment Run-time Model* provide the means through which a reflective behavior may be achieved.

In order to describe a model for the scenario presented in Section 3, we exploit a three layer architectural model (depicted in Figure 4) where (*i*) the *description layer* describes both the application's *functionality* as a workflow containing a sequence of *virtual basic actions* and the *environment* as a set of monitorable *virtual elements*, and (*ii*) the *implementation layer* encapsulates the concrete implementations to which virtual entities may be mapped. Indeed, the *proxy layer* provides a virtualization layer that we use to introduce a further degree of indirection, thus enabling loosely-coupled relations between virtual objects (i.e., basic actions and sensors) and their implementation.

As shown in Figure 4, *description layer* describes both the application model and the environment model. The application model comprises the run-time entities that support the enactment of the behavioral model corresponding to the workflow of the currently active functionality of the set-top box. Since the functionality must be adapted with respect to the actual context, the workflow itself is not directly bound to the concrete implementations of its actions. Rather, dynamic adaptation is achieved by decoupling the virtual basic actions of the workflow at the description level from their concrete implementation, thus achieving the required flexibility supported through dynamic binding.

The *implementation layer* contains the set of all possible components (called implementation components) implementing the virtual basic actions specified within the workflow, as well as other companion components that might be used to support the computation. Each virtual basic action might be implemented by several implementation components that vary from each other in terms of extra-functional properties – e.g., security, performance and reliability. That is, following the Product Line Architecture (PLA) approach [5], the component-based model exploited by both *proxy layer* and *implementation layer* enables components to be specified as "variant", although in our case variant selection is performed at run time. This allows for specifying the alternatives to consider

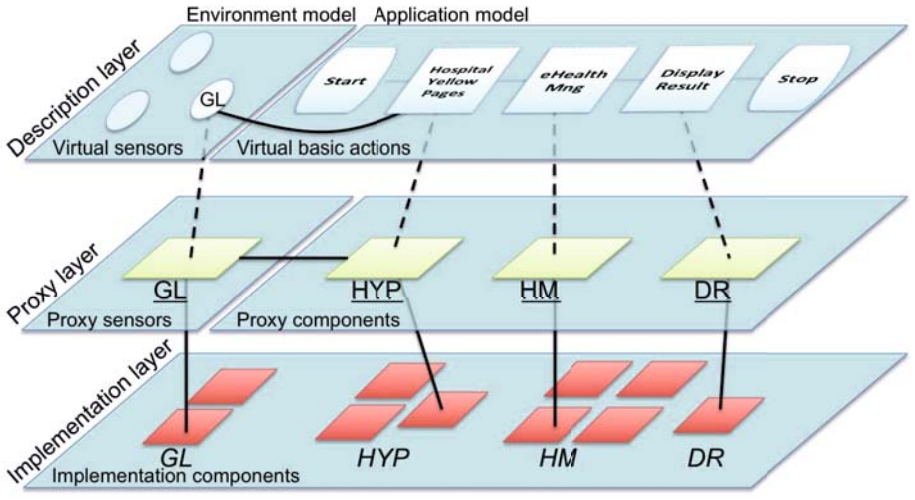


Fig. 4. The three layer run-time model of the application

while mapping virtual basic actions to implementation components. Indeed, alternatives represent the variation points within the run-time model, where the dynamic adaptation of functionality, with respect to the actual needs (i.e., software evolution or context-aware adaptation), can be applied.

In this context, the *proxy layer* plays the role of filtering layer. That is, the components belonging to this layer (called *proxy components*) do not implement the needed functionality themselves. Rather, they implement the logic for variant selection, i.e., they choose among the available components the one that provides the needed functionality and best-fits the requirements – e.g., the most secure, the most efficient, the most reliable. Furthermore, it is worth noticing that no assumption is made about how implementation components are actually implemented. For example, as we will show next, implementation components can be implemented as clients of external and remotely accessible third-party Web Services.

In Figure 4, the *environment model* describes the run-time environment in which the application is executed. In particular, it specifies the set of monitorable *virtual elements* constituting the environment and the data they can provide. The next section describes how such data are made available by means of *proxy sensors* through *implementation sensors*.

5.2 Monitor

Context-aware behaviours and self-reconfiguration require applications to be able to sense the environment and reason about it. Following the meta-architecture presented in Figure 3, the *Monitor* entity is in charge of (i) collecting context data coming from the *Environment* constituents (namely *Sensor* and *Application*) and

(*ii*) updating *Current Application Run-time Model* and *Environment Run-time Model* accordingly.

The *Environment Run-time Model* of Figure 3 maps to *Environment model* of Figure 4, in the *description layer*. This model describes the set of monitorable virtual elements constituting the environment and the data they can provide. As shown in the previous section for virtual basic actions, also virtual sensors are mapped to their corresponding entities in the proxy layer in order to decouple their description from the actual implementations (see Figure 4). That is, each virtual element within the environment is monitored by a proxy sensor belonging to proxy layer.

The workflow specified by the application model in Figure 4, is not environment-agnostic but relies on the set of virtual sensors which constitute the environment model. Indeed, each virtual basic action in the workflow can specify the set of virtual sensors (if needed) to be considered for accomplishing its task. Hence, such a relation must be kept and reflected also at proxy layer where proxy components rely on environmental context data for selecting the specific implementation component to bind with. Hence, each proxy component relates to the proxy sensor needed to gather the environmental context data relevant for selecting the proper implementation component, as well as for computing their tasks.

Furthermore, as done for proxy components, also proxy sensors are implemented as implementation components adhering to the three-layer structure of Figure 4. Also in this case, the separation between proxy sensors and implementation components allows proxy sensors to be implemented by several different implementation components, then allowing them to be dynamically downloaded and deployed when needed.

The next section, describes how such context data, retrieved through *Monitor* and stored within *Environment Run-time Model*, is further queried by *Decision Maker* and used for actuating the application reconfiguration process.

5.3 Decision Maker and Actuator

So far, we have described the entities used by *Decision Maker* to achieve its decisions about run-time reconfigurations, namely those that reify the run-time models. In this section we are going to explain how such decisions are made by *Decision Maker* and further actuated by *Actuator*.

Specifically, referring to the three-layer model described in Section 5.1, decisions that must be made by *Decision Maker* concern the binding (rebinding) between a virtual basic action and the proper implementation component. The proxy component implementing the specific virtual basic action is responsible for this task. The proxy component therefore plays the role of a decision maker. Hence, *Decision Maker* is implemented as the set of *proxy components* deployed at the proxy layer, where each proxy component makes decisions regarding the specific virtual basic action it represents.

While *Decision Maker* retrieves data about the *Environment Run-time Model* through the proxy sensors (see Section 5.2), information about *Application*

Run-time Model are kept by the proxy components themselves and represented as the current binding (i.e., between virtual basic action and implementation component) and the data that have steered the selection of such a binding. That is, every proxy component must know:

- which implementation component has been used during previous executions
- which inputs were received (both by the context and by the previous element in the workflow) that caused that specific binding to be chosen.

The former information is needed to improve performance. In fact, should two identical execution should take place, there would be no binding overhead. The latter information is needed to let proxy components be aware of the fact that a reconfiguration is required. In fact, every change in either the context or the workflow computation will lead to checking if the actual binding is still valid or, otherwise change the binding. Validity of a binding ranges from the mere existence of a target entity that can be reached through the binding, to the fact that the target meets some optimality criteria for quality attributes.

Specifically, a reconfiguration decision might be made to face the following three different cases:

1. A proxy component must select a binding for the corresponding virtual basic action for the first time.
2. A binding between a virtual basic action and its implementation component is no longer valid.
3. The implementation component fails during the execution.

In case no bindings are in place between virtual basic actions and implementation components. The proxy components are in charge of selecting the proper implementation component. The choice is made according to a policy that may take into account extra-functional properties of the available implementation components. In the second case *Decision Maker* must reconfigure the application, to try to still meet the requirements. That is, a new binding must be found in order to accomplish the required task. Finally, in the third case the corresponding proxy component can automatically change the binding using a substitutable implementation component and restarting the computation from scratch for the corresponding virtual basic action. Clearly, if no alternatives are available or all the alternatives fail, the computation specified by the workflow cannot be carried out and an error message is reported to the user.

Following the meta-architecture presented in Section 4, which sharply separates the abstract description of the reconfiguration from its actuation, once the reconfiguration description has been made by *Decision Maker*, it must be passed to the *Actuator*, which in turn will apply it to the current instance of the application. Indeed, in our specific instantiation of the meta-architecture, the proxy component chooses the implementation component that must be used and then passes its reference to the *Actuator* which will perform the following activities:

1. it downloads and deploys the implementation component referenced to the proxy component, if needed.

2. it invokes the implementation component just retrieved by passing the parameters coming from the workflow.
3. it passes back to the workflow the result of the computation coming from the implementation component.

The logic implementing the *Actuator* is provided by the proxy layer itself and is exploited by proxy components every time an invocation must take place.

5.4 OSGi-Based Implementation

Mapping to the proposed reference meta-architecture (see Section 4), in this section we describe how *Application*, *Sensor* and *External Service/Component* entities contained by the *Environment* are actually implemented by means of the OSGi framework.

The scenario presented in Section 3 has been implemented using the OSGi (Open Services Gateway initiative) component-based framework. Many different frameworks have been developed so far for component-based programming such as JavaBeans [18] and COM [16]. Although these systems are widely accepted as standard component-based frameworks, they are not well suited for our purposes, since they do not allow for components addition and removal at run time. More precisely, bindings between components are predefined and fixed, making architectural mutations impossible.

On the contrary, what we need is a framework able to decouple components by achieving a run-time feature that allows both modification of bindings, and components addition and removal. To this extent, the OSGi (Open Services Gateway initiative) [15] is a module system for Java implementing a dynamic component model [4]. At a glance, the core part of OSGi defines (*i*) *bundles* (i.e., components) that can be installed, started, stopped, updated, and removed at run time, (*ii*) the *service registry* that allows bundles to find new services and bind to them, and (*iii*) the *execution environment* that defines methods and classes available within a specific platform (e.g., lower-end device, embedded device, high-end server). As in a service-oriented architecture, an OSGi bundle can publish its services into the *service registry*, making them available to other bundles. The key difference between Web services and OSGi services relies on the fact that while Web services always require some specific transport layer, the OSGi services use direct method invocations. This makes OSGi a well-suited framework for scarce-resource devices.

Referring to the scenario presented in Section 3, Figure 5 depicts a possible implementation of the "eHealth Emergency Management" use case. In particular, the OSGi framework is deployed into the set-top box and contains the set of bundles implementing both proxy components and implementation components that relate to the virtual basic actions specified within the *HEM* workflow (see Figure 2).

Still referring to the meta-architecture presented in Section 4, the *Application Run-time Model* and *Environment Run-time Model* are implemented by means of the three-layer model discussed in Section 5.1, where virtual basic actions

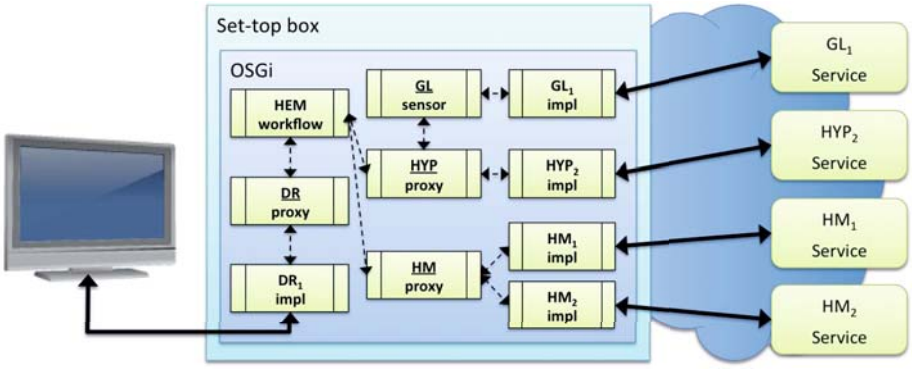


Fig. 5. OSGi implementation of the eHealth Emergency Management System

map straightforwardly to proxy components (i.e., HYP, HM and DR) and their possible implementations, and virtual sensors are monitored by means of proxy sensors. For example, in order to locate the hospital that can better manage the health emergency, the *HYP* virtual basic action needs contextual data about the set-top box geo-location and thus it specifies a dependency with the Geo Location (*GL*) virtual sensor, which, in turn, is implemented as proxy sensor and deployed within the set-top box with the intent of monitoring its position. In particular, referring to Figure 5:

1. A workflow is specified by means of an OSGi bundle (the HEM *workflow bundle*) that invokes the functionalities exposed by the proxy components.
2. A proxy component is implemented by means of an OSGi bundle (the HYP, HM and DR *proxy bundles*), which is responsible for selecting the proper implementation component relying on both extra-functional properties declared by the implementation component and contextual data gathered from proxy sensors.
3. A proxy sensor is implemented by means of an OSGi bundle (the GL *sensor bundle*), which selects the proper implementation component implementing the required monitoring facility.
4. An implementation component is implemented by means of an OSGi bundle (the GL_1, HYP_2, HM_1, HM_2 and DR_1 *impl bundle*), which actually implements the required facility.

In this setting, the HEM workflow is executed as a standard OSGi bundle that invokes the methods exported by HYP, HM and DR proxy bundles, respectively. When invoked, HYP selects the closest hospital by relying on the contextual information provided by GL and returns it to HEM. Clearly GL must be retrieved by the *Service Repository* and installed, unless it is already in place. GL can be in place if another application has already mentioned it as a dependency. After that, GL will search for all the sensors (local or remote) providing the local position and will choose between them according to some policy. To enable HYP to contact GL without the need of hard-coded references, a naming convention

is adopted. That is, proxy sensors have the same name of the corresponding entities in the run-time environment model. In this way the proxy components of every downloaded workflow can easily reference local proxy sensors.

It is important to note that the logic responsible for retrieving contextual data is not application-specific. Rather, once a proxy sensor is downloaded and deployed within the set-top box, it can be accessed and used by many applications at the same time. Once such information has been retrieved, the HYP bundle can select the closest HM hospital and send the alarm to it. The result of such invocation will then displayed on the TV screen through the DR bundle.

As explained above, each *proxy bundle* can select which specific *implementation bundle* to bind to, among the ones that are available. Indeed, this functionality is provided by means of the OSGi late-binding mechanism, which allows for searching, filtering and binding bundles at run time. Specifically, there are three issues regarding such binding selection (refer to Section 5.3): (*i*) a proxy bundle selects the binding to the corresponding implementation bundle for the first time, and (*ii*) the binding between a proxy bundle and its implementation bundle is no longer valid[1]. To face such issues, two strategies are implemented: the former aims at optimizing extra-functional requirements, whereas the second one aims at forcing the application to meet its requirements also in a changed situation.

The first strategy considers extra-functional requirements optimization while binding a virtual basic action to an implementation bundle. For example, when the set-top box must execute the *virtual basic action* corresponding to the *eHealth Manager*, two implementations may be available as possible targets for HM: an *implementation bundle* able to contact the closest hospital's web service or an *implementation bundle* that can contact the hospital via a standard phone call transmitting a pre-recorded vocal message. The two alternatives are functionally equivalent, i.e. delivery of the alarm message to the hospital is guaranteed in both cases. They differ, however, in their extra-functional qualities, such as reliability, performance or cost of connection. A choice can thus be made when the binding is performed according to some optimization strategy. Notice that the same decision making logic can be implemented for the proxy sensors. As an example, the geo-localization proxy sensor could choose between two different implementation bundle, one implemented through a GPS device and the other through an IP-based geo-localization remote service. The two implementation-components may be associated with attributes that specify an accuracy level and a cost; the choice can thus be made by maximizing a quality figure that takes both parameters into account.

On the other hand, the second strategy aims at forcing the application to meet its requirements even in a changed situation. As an example, when the set-top box moved from its original location to a new location, the box's environment changes accordingly. We can also imagine that, during the previous execution of the HEM workflow, the emergency alarm was sent to the closest hospital by

[1] It is worth noticing that, as discussed in Section 5.3, the third issue can be easily solved by reducing it to the first one.

a web service provided by the hospital itself. It may happen that, in the new location, the closest hospital does not provide a web service to handle emergency alarm. It is clear that the emergency alarm must still be delivered, but the way it is delivered must be changed to be adapted to the new environment. Thus the binding established during the previous execution between the eHealth manager virtual basic action and the HM implementation bundle must be changed. Once again, the logic for the adaptation is implemented by a proxy component (HM). In fact, this is the only component in our architecture that knows which is the current binding and which were the inputs from the previous virtual basic action in the workflow, as well as the context that caused such binding establishment. In our use case, given the location change, HM could receive as input a different "closest hospital", and this hospital could support the on-line emergency alarm or not. If the previous communication mode is still supported, then only the web service address must be updated, otherwise a new implementation bundle must be retrieved, downloaded, and deployed to be bound to the eHealth manager virtual basic action.

Notice that a proxy bundle does not directly invoke the selected implementation bundle. Rather, following the meta-architecture an *Actuator* is needed to make the decision application-agnostic. However, in our implementation, the actuator role is played directly by the OSGi framework through the dynamic late-binding mechanism. Indeed, every proxy bundle exploits such a mechanism (that can be logically seen as a proxy layer facility) for selecting and invoking the corresponding implementation bundle, and retrieve the computational results. However, if no implementation bundle matching the requirements is available within the framework, it can be (*i*) downloaded from the TV providers service repository by means of the relative facility provided by the set-top box (see Section 3), (*ii*) published into the OSGi *service registry*, and (*iii*) dynamically invoked at run time.

As few final remarks, concern implementation bundles. They can either implement the required task or be used as a stub to access remote Web Services, which actually implement the tasks. For example, referring to Figure 5, the DR bundle, which depends on the specific TV screen attached to the set-top box, locally implements itself the logic needed to display the workflow outcome on the TV screen. On the other hand, the HM proxy bundle is implemented by a set $\{\mathrm{HM}_1, \mathrm{HM}_2\}$ of implementation bundles which in turn grant the access to HM_1 Service and HM_2 Service, respectively. This solution allows for reducing the computational burden, which is unbearable for scarce-resource devices such as the set-top box, by delegating the effective implementation to the service provider side.

6 Conclusions

Open-world systems are characterized by a highly dynamic software architecture where both components and their interconnections may change dynamically, while applications are running. Pervasive systems are a notable class of open-world systems, where the need for dynamic software architectures are needed to

support the situation-aware behaviors that characterize them, namely *context-aware adaptation* – run-time actions affecting the architectural level which react to environmental changes – and *software evolution* – changes that originate in the requirements.

In this context, software systems must be able to reason about themselves and their state as they operate, through adequate reflective features available at run time. Moreover, they must be able to monitor the environment, compare the data they gather against the expected model, and detect possible situational changes. Whenever a deviation is found, an adaptation step must be performed, which modifies the software architecture.

This paper presented a meta-architecture supporting run-time adaptation and evolution. In particular, it first described the general, high-level reference meta-model by discussing its constituent entities and the relations between them. Then it illustrated how such a meta-architecture can be instantiated and adapted to develop a specific case-study, namely the "eHealth Emergency Management System". Such a scenario describes an adaptive embedded system – i.e., a smart set-top boxes for pay-per-view television – that addresses the functional evolution and adaptation of on-demand software. We finally demonstrated the applicability of such an approach by implementing the "eHealth Emergency Management System" case study through the OSGi component-based framework, which provides a complete and dynamic component-based programming platform for scarce-resource devices. In particular, we detailed how the entities specified by the abstract meta-architecture have been actually implemented within the concrete "eHealth Emergency Management System" application.

It is worth noticing that the specific application domain addressed herein – i.e., pervasive embedded systems – had an impact on the implementation choices we made, since it presents a set of specific extra-functional requirements. The fact of dealing with resource-scarce devices asks for implementations to be as light as possible, while still satisfying the functional requirements. This is the reason why the abstract application workflow, which might naturally specified by means of a high-level interpreted language (e.g., XML-based), has instead been encoded directly into Java, to reduce run-time overhead. This in fact speeds up execution and eases deployment within the framework by supporting dynamic installation and removal of applications.

Pervasive embedded systems also have to be dependable. An applications like the "eHealth Emergency Management System" must provide an acceptable level service quality even in critical situations. This imposes certain requirements on the *Decision Maker*, which should be able to dynamically reconfigure the bindings to external components and services to achieve the required self-healing capabilities. The proposed solution also ensures a level of security and trust because *implementation components* are provided through a centralized and controlled repository. In a more general case, it might be useful to extend reconfiguration policies beyond just re-binding, e.g. also supporting re-plan of the workflow on-the-fly.

The work described in this paper is part of an on-going long-term research effort that focuses on self-managing situational software systems. Part of this work addresses software architectures that best match the goals of such systems. One possible outcome of this particular work would be the identification of a catalog of architectural solutions that may be adopted in different systems, which would fit the specific characteristics of the systems under consideration. To gain precise, reliable, and reusable knowledge about the various solutions, we are currently engaged in different case studies, ranging from decentralized distributed systems supporting urban mobility scenarios to emergency-management to rescue people in mountain areas. Different architectural solutions will be defined and tried in the case studies, to gain a deep understanding of their potential benefits and drawbacks, and eventually support the development of the solutions catalog mentioned above.

Acknowledgements

This research has been Funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom (http://www.erc-smscom.org).

References

1. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. Computer 39(10), 36–43 (2006)
2. Baresi, L., Guinea, S.: A-3: Enabling self-adaptation in distributed systems through group abstraction. Technical report, Politecnico di Milano (2009)
3. Bindelli, S., Nitto, E.D., Mirandola, R., Tedesco, R.: Building autonomic components: The selflets approach. In: 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops, ASE Workshops 2008, pp. 17–24 (2008)
4. Cervantes, H., Favre, J.-M.: Comparing javabeans and osgi towards an integration of two complementary component models. In: Proceedings of EUROMICRO Conference (2002)
5. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley Pub. Co., Reading (August 2001)
6. Erenkrantz, J.R., Gorlick, M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of web architectures. In: ESEC-FSE 2007: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 255–264. ACM, New York (2007)
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
8. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. 2(2), 115–150 (2002)
9. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
10. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer, 41–50 (2003)

11. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. 16(11), 1293–1306 (1990)
12. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., et al.: OWL-S: Semantic markup for web services (2004)
13. Nitto, E.D., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Automated Software Engg. 15(3-4), 313–341 (2008)
14. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE 1998: Proceedings of the 20th International Conference on Software Engineering, Washington, DC, USA, pp. 177–186. IEEE Computer Society, Los Alamitos (1998)
15. OSGi Alliance. OSGi service platform, core specification, release 4 (2007)
16. Platt, D.S.: Understanding COM+. Microsoft Press, Redmond (1999)
17. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Englewood Cliffs (April 1996)
18. Sun Microsystems, Inc. JavaBeans, http://java.sun.com/products/javabeans
19. Sykes, D., Heaven, W., Magee, J., Kramer, J.: Plan-directed architectural change for autonomous systems. In: SAVCBS 2007: Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems, pp. 15–21. ACM, New York (2007)
20. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. In: SEAMS 2008: Softw. eng. for adaptive and self-managing systems, pp. 1–8. ACM, New York (2008)
21. Taylor, R.N., Medvidovic, N., Oreizy, P.: Architectural styles for runtime software adaptation. In: WICSA/ECSA 2009 (2009)
22. Traverso, P., Pistore, M.: Automated composition of semantic web services into executable processes. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 380–394. Springer, Heidelberg (2004)
23. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Softw. Eng. 33(12), 856–868 (2007)

Using Graph Grammars for Modeling Wiring Harnesses – An Experience Report

Albert Zündorf[1], Leif Geiger[1], Ralf Gemmerich[1], Ruben Jubeh[1], Jürgen Leohold[2], Dieter Müller[3], Carsten Reckord[1], Christian Schneider[1], and Sven Semmelrodt[4]

[1] University of Kassel
[2] Volkswagen AG
[3] Sumitomo
[4] Siemens VDO Automotive AG

Abstract. The Fujaba project has created a light weight graph grammar approach allowing the use of graph grammar concepts in usual Java programs. Fujaba comes with visual editors for graph schemas / class diagrams, control structures / activity diagrams, and graph rewrite rules / extended object diagrams. Thereby the user may specify executable programs that are translated into complete Java programs without any need for further low level Java programming. In addition, Fujaba provides dedicated visual language support for scenario based testing. This is complemented with support for model versioning and distributed applications. Last but not least Fujaba provides model level debugging.

This paper is an experience report applying Fujaba techniques for requirements analysis and implementation in an industrial project in the automotive industry. The considered project has created a tool for the design of car electronic systems. This project involved an enormous amount of domain knowledge. The challenge was to involve the domain experts in the analysis, design, and implementation activities such that the transfer of domain knowledge is fostered. This paper reports how we used graph grammar based Fujaba techniques and languages to achieve domain expert involvement.

1 Introduction

Modern software development processes, as e.g. the Rational Unified Process [JBR99], use textual use case descriptions for requirements elicitation. During further requirements analysis, analysis classes are identified and one may e.g. use collaboration diagrams or sequence diagrams in order to outline use case behavior. This is then further refined into design and implementation classes and into actual behavior implementation. This process may be executed iteratively, addressing small chunks of functionality one after the other.

Due to our experiences, such a process requires decent skills in abstraction, object orientation, analysis, design, design patterns, etc. Frequently, the result of an object oriented analysis or design phase will be documented and discussed

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 512–532, 2010.

using architecture diagrams or more specifically class diagrams. Unfortunately, class diagrams are quite abstract and their interpretation and the judgment of the represented design decisions requires special skills and training, too. While many computer scientists and professional software developers have these skills as part of their every day professional work, a project frequently involves a large number of other stakeholders like business managers, administrative experts, domain experts and potential system users. These people may have quite different backgrounds and skills. These different skills and the corresponding domain knowledge are of crucial importance for the success of the overall project. Unfortunately, many of these important people will have difficulties to interpret the information and design decisions contained in architecture and class diagrams. Thus, the domain experts are frequently not able to communicate their knowledge with the software experts, appropriately.

In order to overcome these communication problems, the Fujaba Process proposes to use object diagrams or graphs or graph scenarios as an intermediate representation or analysis means during analysis, design and implementation, cf. [DGZ05, KNNZ00, FNTZ99]. Usual collaboration or sequence diagrams stay on a level of abstraction where certain components exchange certain messages in order to exemplify the internal process that realizes the considered use case, cf. [JBR99, BRJ99]. Our graph scenarios usually describe the internal process in much more detail. We employ concrete objects with concrete attribute values and concrete links or edges between each other. Thereby, we pinpoint the representation of certain domain information in our implementation model quite early in the development process.

In the reported project, it turned out, that the design and implementation decisions outlined in graph scenarios were still easily recognized by our domain experts. Pinning down certain design and implementation aspects quite early then fertilized the further refinement and elicitation of related functional requirements, considerably. In multiple iterations, our domain experts used the refined designs to enhance their textual requirements descriptions until it became easy to implement them. As implementation language, the Fujaba approach uses programmed graph transformations, so-called Fujaba story diagrams. Since story diagrams use a graphical notation which is very close to our graph scenario notation, our domain experts were even able to review our implementation. This enabled very valuable feedback from our domain experts to the software developers. For the goals of the reported project cf. [GSZ et al. 05].

Note, this paper is an extended version of [ZLM et al. 06]. First, we have added more details. Second, we have added an explicit description of the Fujaba process. In addition, we made the relation to graph grammars more explicit.

The following section gives a short overview of the Fujaba process and relates graph grammar terminology to object oriented terminology. Section 3 introduces the application domain and the software development project that serves as example in this paper. This is followed by the main section 4 showing how we applied our software development approach to the example application. After a

short discussion of related modeling languages the last section concludes with lessons learned.

2 The Fujaba Process FUP

The Fujaba process (FUP) is a light weight, iterative, use case driven process deploying graph grammar techniques for the development of object oriented applications. For each feature or functionality of the desired software, the Fujaba process proposes the following steps, cf. [DGZ05]:

Step 1: Textual scenarios

Once a feature has been identified, the desired functionality is outlined with the help of scenario descriptions in natural text. Such a scenario focuses on one concrete situation and on the steps that should be executed in this dedicated situation. To facilitate this step, we stay at the domain level, i.e. we do not yet map the application domain to the implementation domain.

Step 2: Graph / model level scenarios: story boards

For each scenario step, we now model an example graph (object structure) that reflects the relevant aspects of the described situation and we model the modifications of that graph that reflect the execution of the scenario step. We call a textual scenario with object models for each step a *story board*. A story board maps domain concepts to the domain of graphs or object structures thus achieving a large step towards implementation. However, to facilitate this modeling step, we stay at the level of example situations and we exploit that the textual scenarios have already been split into a number of smaller steps which are now easier to model.

Note, due to our experiences, since our graph scenarios stay at the level of concrete example objects, even non IT experts are able to read and understand them, easily. However, to facilitate the recognition of graphs / object diagrams for non IT domain experts even more, we frequently use icons from the application domain to enrich the representation of objects. The domain experts grasp the semantics of such domain icons immediately.

Step 3: Class diagram adaption

Deriving a graph schema or a class diagram from an object diagram is a straight forward task. The result is a conceptual model / class diagram that covers the identified domain concepts. Usually, this conceptual model will be refined towards implementation aspects. For example, there will be extensions for implementation concepts and refactorings for the introduction of design patterns.

Note, usually we do not bother non IT domain experts with these class diagrams. They are mainly of use for the software developers.

Step 4: Graph or model implementation

Our approach does not deploy a generic graph implementation or a generic graph database. Instead, our Fujaba tool provides a code generator that implements models using plain old Java objects/classes. However, our code generator uses a dedicated programming style for the implementation of bidirectional associations. This association implementation guarantees referential integrity while

still providing an excellent runtime efficiency. This referential integrity turns the used object structures into graphs. Alternatively, we also generate EMF ([EMF]) based model implementations. This enables the use of a large number of EMF based tools for our graphs.

Step 5: Test derivation

Once our class diagrams are implemented, our code generator turns the story boards into automatic JUnit tests, cf. [GZ05]. Basically, such a test creates the object structure modeling the start situation, triggers the described user actions and validates that the object structure shows the modifications described in the story board as the effects of the executed operations.

Step 6: Textual algorithm design

Now we are ready to design the desired functionality. To facilitate this complex development task, we start with the design of an appropriate algorithm on the domain level using natural text.

Step 7: Model level algorithm design

Once we have outlined the general algorithm, we go through its steps and refine them towards the level of our object model. This means, domain level sentences are refined to refer to model elements.

At this step, we leverage that the general algorithm outline has already been done. Thus, we deal with one algorithm step after the other which facilitates this step, considerably.

Step 8: Implementation

The model level algorithm description is now implemented. While this could be done directly in Java, our approach proposes to use programmed graph rewriting for this approach. In Fujaba, we use activity diagrams for the modeling of control flow and enriched objects diagrams as notation for graph transformations. Our code generator translates this to Java code either based on our own simple model implementation or based on an EMF model implementation.

Step 9: Algorithm validation

Now we validate our implementation by running the JUnit tests derived from the story boards in step 5. Although this does not provide a systematic whitebox test, due to our experiences the scenario based tests achieve already a reasonable coverage of the implementation. If occasionally certain parts of the implementation are not covered by scenario tests, most of the time the programmer has already implemented alternative behavior which is not yet addressed by the provided scenarios. Our approach takes such uncovered implementation as an advice to build additional scenarios covering this not yet addressed behavior and to go back to the domain experts to discuss whether the implemented behavior for these special cases meets their expectations.

Step 10: Refactoring

Once the functionality has been validated, we may perform some structuring of our code and some refactorings as recommended in agile processes.

Note, we use the Fujaba process since several years in our research projects and for the development of new features of the Fujaba tool itself. We also teach the Fujaba process and its underlying graph based techniques since 2003 in our

courses on programming methodologies at Kassel University. All these experiences resulted in many enhancements and refinements of the Fujaba process until we reached the state described above. While our experiences with the Fujaba process are very encouraging, its enhancement and the improvement of its tool support are ongoing work. However, one strength of the Fujaba process and its technologies is the high level of abstraction reached by the underlying graph concepts. Together with the scenario oriented approach, this facilitates the involvement of non IT domain experts, considerably. A striking example of a successful involvement of domain experts in the development of a complex expert system for the optimization of wiring harnesses is outlined in the remainder of this paper.

Fujaba and the Fujaba process have been developed in the context of graph grammar theory. However, Fujaba and the Fujaba process are frequently used for object oriented modeling and implementation. In the domain of graph grammars, we use graph grammar terms like, node, edge, or graph. In the domain of object oriented modeling and implementation, we use OO terms like object, link, or object structure instead. Table 1 shows the relation between graph grammar terms and OO terms. Through the rest of this paper, we will use the OO terms.

3 The OBA Application

The considered example project is called OBA for "Optimization of car electric system architectures" (in German: "Optimierung von Bordnetz-Architekturen"). Our task was to develop a tool that enables a car manufacturer to optimize the costs of the electronic system within a car. The considered electronic system of a car includes actors (e.g. lights, motors, ...), sensors (e.g. on/off switches, temperature sensors, rotation sensors, ...), electronic control units (ECUs), fuses, and all the wires connecting these components. The costs of the electronic system of a car consist of the costs for the parts plus the costs for assembling them within the car plus the costs of extra material as e.g. cable ties or screws, etc.

Prerequisites for the development of a car electronic system are usually the employed sensors and actors including their placement within the car. In addition, the geometry of the car body is usually already defined including possible spaces for the placement of ECUs and including all channels where cables may be laid and mounted.

The placement of ECUs and the laying of cables has to respect a large number of additional constraints, e.g. some components must not be accessible from the outside (e.g. anti theft devices), some components should not get wet, some components should not get too hot (e.g. near the engine), some components may interfere with each other (electromagnetic compatibility). There are also robustness and security constraints, e.g. the cable diameters have to be designed according to the connected power consumption and fuses have to be designed to enable the desired power consumption but to prevent cable fire, the driving lights must not depend on a single fuse, etc.

Table 1. Graph terms vs. object oriented terms

Graph Term	OO Term	Remark
node	object	
edge	link or pointer	Generally, edges are bidirectional. To achieve this in OO programming languages like Java, Fujaba uses pairs of forward an backward pointers.
graph	object structure	A set of objects related by links between these objects.
graph	object diagram	We distinguish between runtime object structures and (UML) object diagrams, which are drawings of object structures used in documentation or in Fujaba for programming.
graph scenario	story board	Sequence of object diagrams outlining the evolution of an object structure during the execution of an operation.
graph schema	class diagram	Beyond graph schemas, class diagrams may also declare methods.
graph rewrite rule	story pattern	In Fujaba, a story pattern is an object diagram extended by modification annotations. At runtime, the story pattern is matched against an appropriate part of the runtime object structure. The matched elements are then modified as specified by the modification annotations.
programmed graph transformation	story diagram	In Fujaba, story diagrams are UML activity diagrams where the activities hold story patterns. Story diagrams specify the control flow between story patterns. Story diagrams are attached to methods and thus provide the method implementations.

Fig. 1. An example car electric system

The main architectural decisions for a car electric system are how many ECUs it employs, where these ECUs are placed in the car and which control functionality is placed on which ECU. Similarly, the number and placement of fuse boxes have to be chosen. Depending on these decisions, the wiring of the car has to be routed to connect the ECUs with each other and with the sensors and actors they need to access and to provide the power supply to all components with minimal costs but respecting the car geometry and the corresponding electrical constraints. Our task was to develop tool support for the developers of car electronic systems enabling them to study alternative architectures with minimal effort and helping them to optimize the electronic system with respect to overall costs and e.g. weight.

As a prerequisite for the optimization of a car electric system, in a first project phase we have developed a cost model that allows us to compute the overall costs of a chosen electric system and to analyze the gain or loss caused by an alternative architecture. In order to facilitate the development of alternative architectures, the second project phase has developed support for labor-intensive work as the configuration of ECUs and the routing for the wiring harness once the main architectural decisions have been done. In a third project phase, the OBA tool shall automatically generate and compare alternative architectures. In this paper we utilize the cost model to illustrate how Fujaba enabled us to involve the domain experts. This means, this paper deals mainly with the first project phase.

The cost model requires an enormous amount of domain specific knowledge. This knowledge is provided by the industrial partners in this project, namely the Volkswagen AG, a car manufacturer or OEM, Siemens VDO, a supplier of ECUs and electronic components, and VW Bordnetze, a manufacturer for wiring harnesses. In addition, there were a PhD student and a post-doc in electrical engineering from the University of Kassel involved, who had the task to transfer the domain knowledge into the project. These engineers were part time supported by the professor for car electric systems at University of Kassel. The software engineering and software development part of the project was executed by two PhD students in computer science together with some student programmers part time supported by the professor for software engineering at University of Kassel.

4 The OBA Cost Model

Since we employ complex data structures for the optimization of the wiring harness and for the conception of ECUs, and due to our experiences, we decided to develop the OBA tool with an object oriented model using the Fujaba environment [Fu09, FNTZ99] and to generate the actual implementation in Java.

The modeling of the OBA application started with the car body. On top of this we have build the wiring model and the models for the ECUs and for the software functions. This is the basis for the optimization algorithms.

In the OBA project one major source of complexity was the sheer amount of aspects that had to be addressed. For example the costs for the wiring harness included aspects like:

- Cable costs depending on cable length and cable kind.
- Costs of cable mounting elements depending on the crossed car areas and their respective mechanical stress. Some car areas may require additional mounting elements including the respective costs.
- Costs of wire protection means. For example, different (humid) car areas require different wrapping techniques with different tape material. Since tape may be as expensive as wires, we had to compute the length of the required tape, exactly.
- Costs and kinds of connectors. For each bundle of wires reaching an ECU or an actor or a sensor, we had to compute an appropriate connector. Depending on the cross section of the different wires, pins of appropriate sizes have to be used. However, the whole connector must not exceed a certain size e.g. the size of the ECU itself or the size of hatches where the bundle has to be threaded through. In addition, for a connector with a high number of pins, the physical force required to put the connector in the socket must not exceed a certain maximum.
- Cables may be split at a certain point in order to reach multiple target points with the same signal. This is achieved using so-called splices. However, there are a number of restrictions on the fan-out of such a splice and on the environmental conditions of the car area where this splice is placed.
- The costs of mounting the wiring harness depends on the accessibility of the corresponding car areas and on the kind and number of mounting elements that are used and on the number of connectors that have to be placed.
- So called threading costs. If some cable bundle needs to be threaded through some car body hatch or some rubber spout, this requires mounting costs according to the number of connectors that are to be threaded and according to the so-called threading length.

This list is far from completeness. Covering ECUs, fuse boxes, communication busses etc. involved similar lists. One of the main challenges of the OBA project was to structure all of these aspects appropriately. This required a close collaboration between the domain experts and the software engineers.

4.1 Applying FUP Steps 1 and 2

As proposed by the Fujaba process, we started the elaboration of required functionality with the help of textual scenarios and object diagram scenarios.

At the beginning of the development, our domain experts came up with textual requirements descriptions as shown in Figure 2. These requirements where of course quite coarse grain and thus insufficient for further analysis, design and implementation. In addition, most other requirements descriptions were based on the car body description, e.g. the description of spatial constraints for ECUs

or for the diameter of cable bundles. The next layer of requirements was based on these two lower requirements layers and so on. Without a clear common understanding of the basic requirement layers, it became progressively harder to read and write new requirements on top of them.

2.1 The car body

…
The car body consists of different car areas representing units of assembly and units of similar environmental conditions. The changeovers between car areas require additional protection means against different environmental conditions, e.g. humidity. In addition, car area changeovers require special mounting efforts. …The car areas contain all kinds of mutually connected wire channels, cavities, etc.
…

Fig. 2. First textual requirements on car bodies

To overcome these problems, we had to pinpoint the requirement layers such that a common understanding between domain experts and software engineers was achieved and such that the next layer was easy to build on top of them.

According to the FUP, we asked our domain experts to exemplify some typical situations with the help of object diagrams. After some training and some more discussion, we developed object diagrams as shown in Figure 3. Actually, the object diagram shown in Figure 3 has a large number of predecessors that were refined multiple times for each requirements layer. We started with object diagrams modeling car areas in order to deal with environmental conditions as hot and wet in the engine area or cold and dry in the cabin area, cf. objects `ca1` and `ca2` at the top of Figure 3 (attributes are omitted due to space restrictions). Then we added channels and nodes modeling the parts of the car body where wires may be routed, cf. objects `ch1` through `ch4` and node `n1`. In addition, the information on required wire protection means and required mounting means is attached to channels (not shown in Figure 3 due to space restrictions). Next we modeled ECUs and other electronic components, cf. `ecu1` and `comp1` and `comp2`. Based on this, we introduced wires, contacts and connectors. Finally, we introduced bundles of wires since some channels may contain multiple wire bundles where each bundle has its own costs for protection means and mounting means. Altogether this process resulted in a large number of detailed object diagrams exemplifying our modeling of the different aspects to be considered for our wiring harness cost model.

With the help of the object diagrams, we had a lot of fruitful discussions with our domain experts. For example, in a first attempt our domain experts proposed that each channel object has its own attributes for the humidity and for the maximum temperature. We recognized, that many channel objects had the same values for these attributes and asked the domain experts, when these values will differ. They told us, that these values are determined by the car area and they differ only when the car area is changed. Thus, we moved the humidity

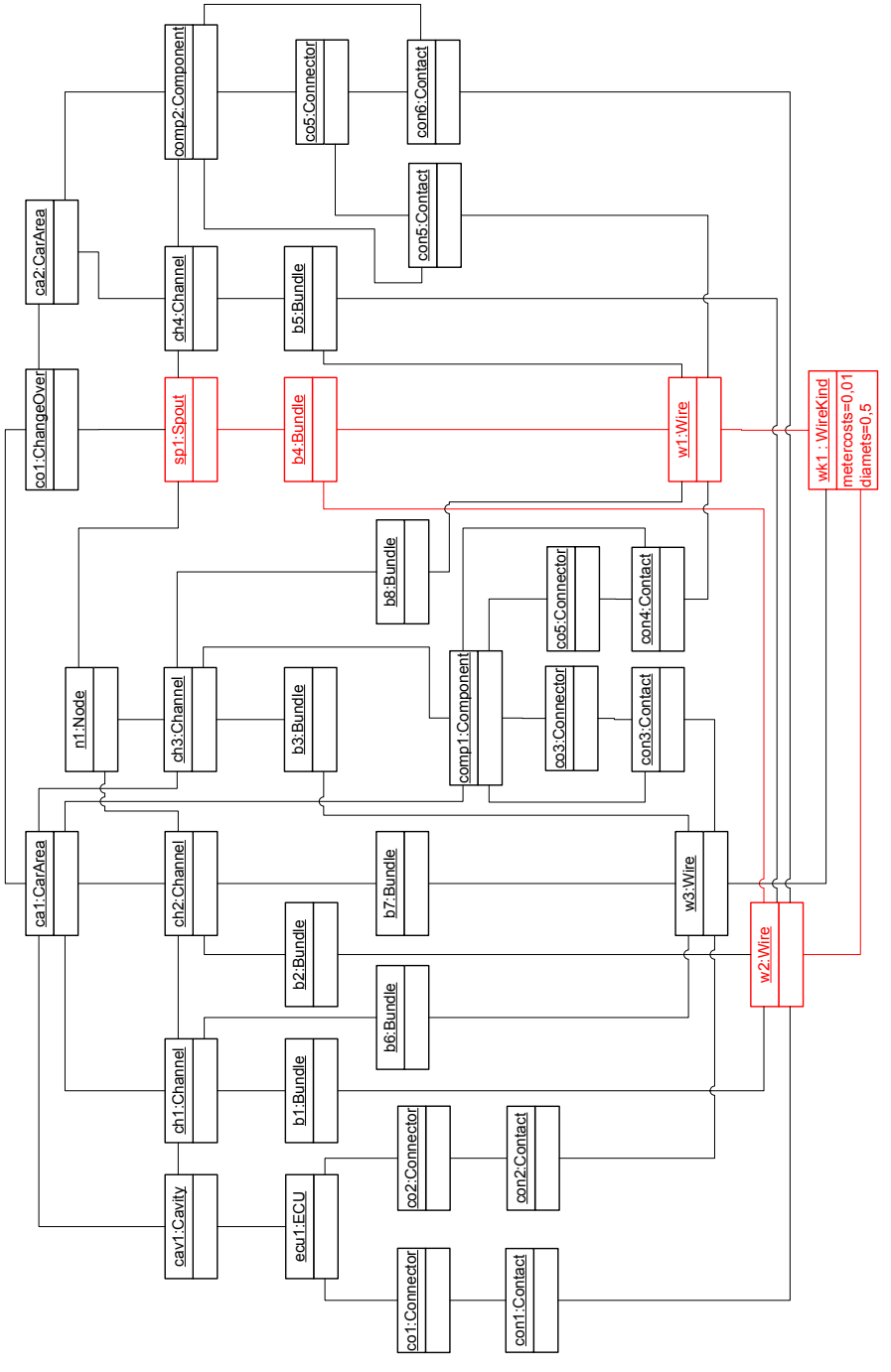


Fig. 3. An object diagram exemplifying the threading costs

and temperature attributes from the channels to the containing car areas thereby reducing the data redundancy a lot.

Similarly, at the beginning, each wire had its own attributes for meter costs, diameter and wire type. After some while, we found out that only 7 different wire diameters are used in a car and that the costs per meter were the same for all wires with the same diameter and the same wire type. To improve this, we introduced a catalogue of wire kind objects holding the price information and each wire just refers to its wire kind. Thereby, it became easy to handle changing market prices for wires. Later on, we introduced such catalogue objects also for contact kinds, connector kinds, chip kinds, etc.

Another important point was the discussion of associations. We did not ask our domain experts whether class `CarArea` should have a `contains` association to class `Channel` or whether we should introduce a composite pattern at this place. Due to our experiences, it is not fruitful to discuss with non software domain experts on this level of abstraction. Instead, we asked our domain experts, whether there is an example of deeper nesting of car areas. Indeed, some car areas contain sub car areas. In addition, there were examples where car areas contained cavities which in turn contained channels. Thus, we decided to use a composite pattern to model this hierarchy and told our domain experts that arbitrary nesting of car areas, cavities, and channels are enabled, now. Note, this design decision has been elaborated during the derivation of class diagrams corresponding to step 3 of the Fujaba process. Thus, it would belong to the next subsection. However, while the software experts used class diagrams to identify the problem, the team used object diagrams to discuss the problem with the domain experts. This corresponds to step 2 of the Fujaba process and thus it belongs to the current subsection, too. Actually, these steps may be executed intertwined.

Overall, the object diagrams provided the domain experts with a clear understanding how the software models the different aspects of the application domain. This also facilitated the textual description of the requirements, considerably. As an example for a more complex requirement description developed this way, we now discuss the computation of threading costs for cable bundles. Figure 4 shows a small example for a wiring harness connecting three components where the right component is reached via a car area changeover that is protected by a rubber spout. To mount the wiring harness, some cable bundle needs to be threaded through the spout. This is a very tedious work with high mounting costs.

Originally, the computation of these threading costs was quite unclear and hard to explain, cf. Figure 5. This first textual requirements description was hard to implement since it was quite unclear, how one finds the wires on the two sides of a spout and how the length is computed. With the help of example object diagrams, it became clear, that one has to consider the bundles within the spout. In addition, it became clear that there are two cases to be considered. If the spout leads directly to a channel, one may just lookup the bundles in that channel. If the spout leads to a node, one has to consider all channels attached

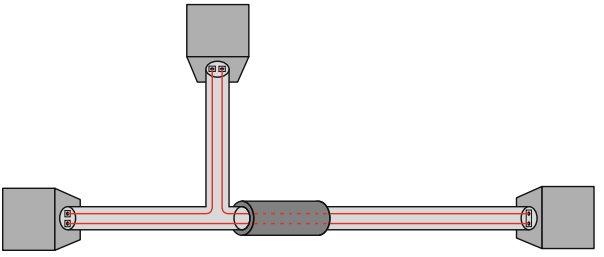


Fig. 4. Example for the threading of cable bundles

to the node. In the project, the graphs were just discussed with the software developers. In this exceptional case, we lazily did not write down the improved requirements description that resulted from the discussion. However, for this paper, we created an improved textual requirements description referring to the graph of Figure 3 in order to exemplify how the graphs helped to improve the requirements description, cf. Figure 6.

Computing Threading Length and Threading Direction
Consider each wire passing a rubber spout and sum up all lengths of all used cable channels before the spout. The distance of spouts and cavities is computed via the lengths of the cable channels. Consider only cavities connected by wires threaded through the spout. Compare the maximal lengths for each side of the spout. The lower one is the required threading length. This also identifies the threading side, i.e the side from which the cable bundle is threaded through the spout. Consider also the number of plugs at the threading side.

Fig. 5. First iteration requirement description

To summarize, with the help of object diagrams the requirements became very detailed and concrete. Such detailed requirements accompanied with elaborated object diagram examples were an excellent basis for the implementation of the desired functionality. Thus, the object diagrams enabled our domain experts to contribute very valuable aids to the design and implementation of the system.

4.2 Applying FUP Steps 3 and 4

Usually, architecture and class diagrams play an important role in software projects. Commonly, these diagrams are used for design discussions. In the OBA project, most of the design decisions have been done based on the discussion of object diagrams. At the first glance, class diagrams have been derived from these object diagrams by just collecting the employed classes, attributes and

Threading Length

- If a bundle of wires, cf. bundle `b4` in Figure 4, needs to be threaded through a spout `sp1`, this requires mounting time according to the number of connectors that need to be threaded and to the length of the bundle that needs to be threaded.
- To compute the *threading length*, both sides of the spout have to be considered, in our example channel `ch4` and node `n1` are *direct neighbors* of spout `sp1`.
- If the direct neighbor is a channel, we consider the wires belonging to bundles that belong to the spout and to the neighbor channel. In our example, channel `ch4` contains bundle `b5` containing wires `w1` and `w2`.
- If the direct neighbor is a node, we consider the neighbor channels connected to that node. Only, wires belonging to the spout and to one of these neighbor channels are considered. In our example, this are again wires `w1` and `w2`. Note, wire `w3` must not be considered since it does not cross the spout.
- The length of a wire is the sum of the lengths on channels containing bundles that contain the wire.
- The maximal length of the two sides of the spout is the threading length.

Fig. 6. Improved requirement description based on object diagrams (Created to exemplify object diagram discussions)

associations. Actually, during the editing of object diagrams, we used class diagrams mainly as a glossary. This means, if one creates a new object, he decides whether the type of this object is already known and may be reused or whether a new object kind, i.e. a new class, needs to be introduced. Similarly, we handled the reuse of attribute declarations and associations. Thereby, the class diagrams leveraged the consistent use of classes, attributes and links through the different object diagrams.

However, inheritance structures and design patterns were introduced at class diagram level. This was the task of the software experts. Discussing these class diagrams with the domain experts was not fruitful.

As an example, Figure 7 shows some central classes of the OBA model. As already discussed, OBA needed a flexible hierarchy of car body elements. This was enabled by a composite pattern created through the class `HierarchyElement` and its `contains` association. In addition, OBA employs several graph like substructures. For example, channels, nodes, and cavities form a graph for the layout of the wiring harness. Similarly, wires, contacts, connectors, ECUs, and components form a graph. These graphs require a certain flexibility since e.g. at the beginning wires are directly connected to components and later on contacts and connectors are introduced. To achieve the required flexibility, we introduced the `at` association for class `HierarchyElement`. This `at` association allows arbitrary graph connections between car elements, even between car body elements and wiring elements which is more then we wanted to achieve. A better modelling would have employed UML 2.0 association refinement features in order to separate between different graph structures. While we wanted to employ these UML 2.0 features, we had no sufficient tool support for the generation of the

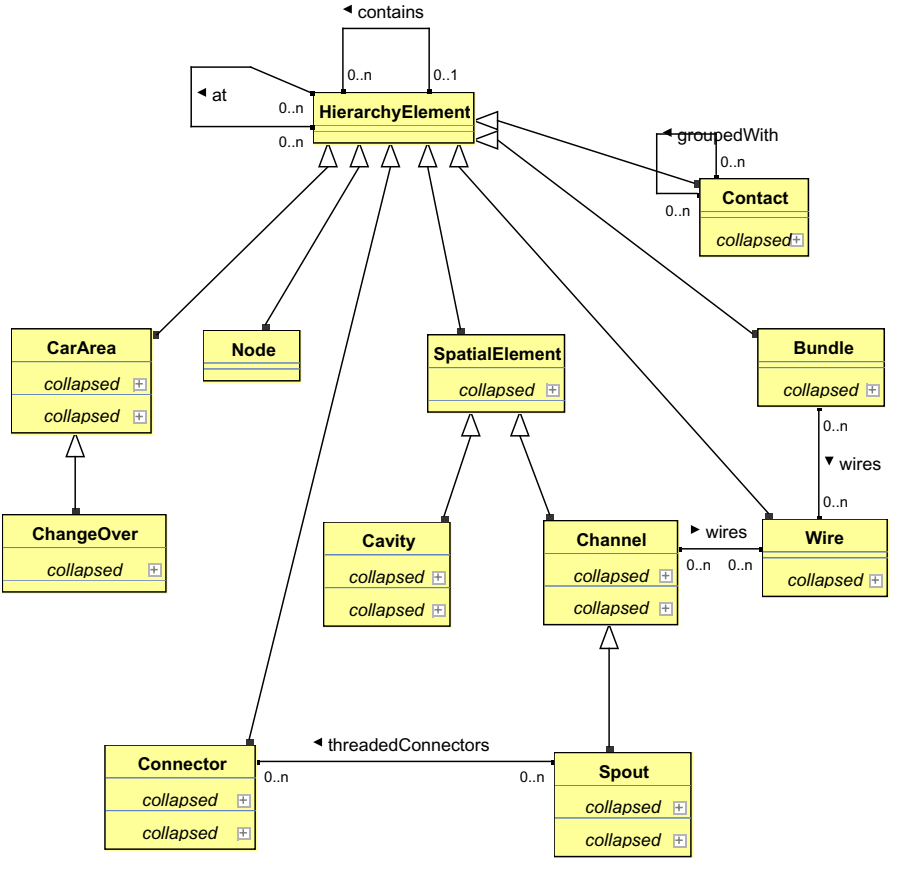


Fig. 7. Some central OBA classes

implementation of such refined associations available and thus we stick to the general solution.

Conventional approaches tend to express as many requirements and constraints in the class diagram as possible. Such conventional approaches frequently use additional OCL expressions to statically constrain the class diagram even more. If class diagrams are your central document, you need to put all your findings there. Contrarily, the Fujaba process proposes to document the central findings in story boards. Thus, class diagrams become secondary documents and as in the above example the class diagrams used in the Fujaba process are usually less restrictive than conventionally. Due to our experiences, such less restrictive class diagrams and designs turn out to be more flexible in case of new requirements showing up in later phases. In our example, the general `at` association allows to connect wires directly to actors while in reality the wires end in a connector that then is plugged to the actor. However, during implementation, it became handy to connect wires to actors first and to insert connectors later

after all wires had been added. In our relaxed design this intermediate object structure was no problem. A conventional design may have easily forbidden it.

4.3 Applying FUP Steps 5 through 8

Once the major design for some functionality had been developed based on detailed textual requirements, object diagrams and class diagrams, we implemented that piece of functionality in an iterative process. Since this paper focuses on the usage of object diagrams and story boards for the involvement of domain experts, we do not discuss the algorithm development here. However, as implementation language we used so-called *story diagrams*, i.e. a combination of activity diagrams and a variant of object diagrams, so-called *story patterns*, cf. [KNNZ00, DGZ05, Zün02]. As an example, Figure 8 shows the specification of method `getThreadingCostsPerCarArea` of class `Spout`. The activity diagram part of a story diagram specifies the control flow as within a flow diagram. Story patterns specify a query to the object structure in the program's main memory. For such a story pattern, the Fujaba environment generates usual Java code. This Java code searches for a match of the depicted object diagram and if this is successful, the depicted operations are executed, cf. [Zün02]. For example, the first story pattern of Figure 8 specifies a query for three objects `this`, `obasystem`, and `params` that are connected by an `allObaElems` link and by a `parameters` link, as depicted. Query objects that are shown without a type are so-called *bound objects*, i.e. they are already known, e.g. the `this` object. The Java code generated for such a query starts from known objects and uses getters and iterators to look-up the other query objects. If these look-up operations fail to retrieve valid matches for all participating objects, the story pattern execution fails. Story diagrams provide special transitions in order to react on the failure or success of a story pattern, cf. Figure 8.

The middle story pattern of Figure 8 looks-up all wires belonging to two bundles that belong to the `this` object and to the (`channel`)`neighbor` object, respectively. For each match[1], the depicted collaboration message computes the threading length for that wire and the maximal threading length is collected in variable `threadingLength`. Similarly, the last but one story pattern handles the case that the neighbor of the current spout is a node and all channels attached to that node have to be considered.

The point is, that our method implementation with story diagrams use a graphical notation (cf. [KNNZ00, DGZ05, Zün02]) that again employs a variant of object diagrams. Since object diagrams have already become familiar to our domain experts during the requirements analysis, with some training, our domain experts were able to review our implementation. We had a lot of fruitful discussions, where the software developers explained their story diagrams and then the domain experts started to point to special cases. In these discussions, the domain experts again used object diagrams to illustrate the special cases they were concerned about. Then the software developers used simple walk throughs

[1] Potentially, a story pattern query may deliver multiple matches. For-each activities depicted by two stacked activity shapes allow to iterate through all such matches.

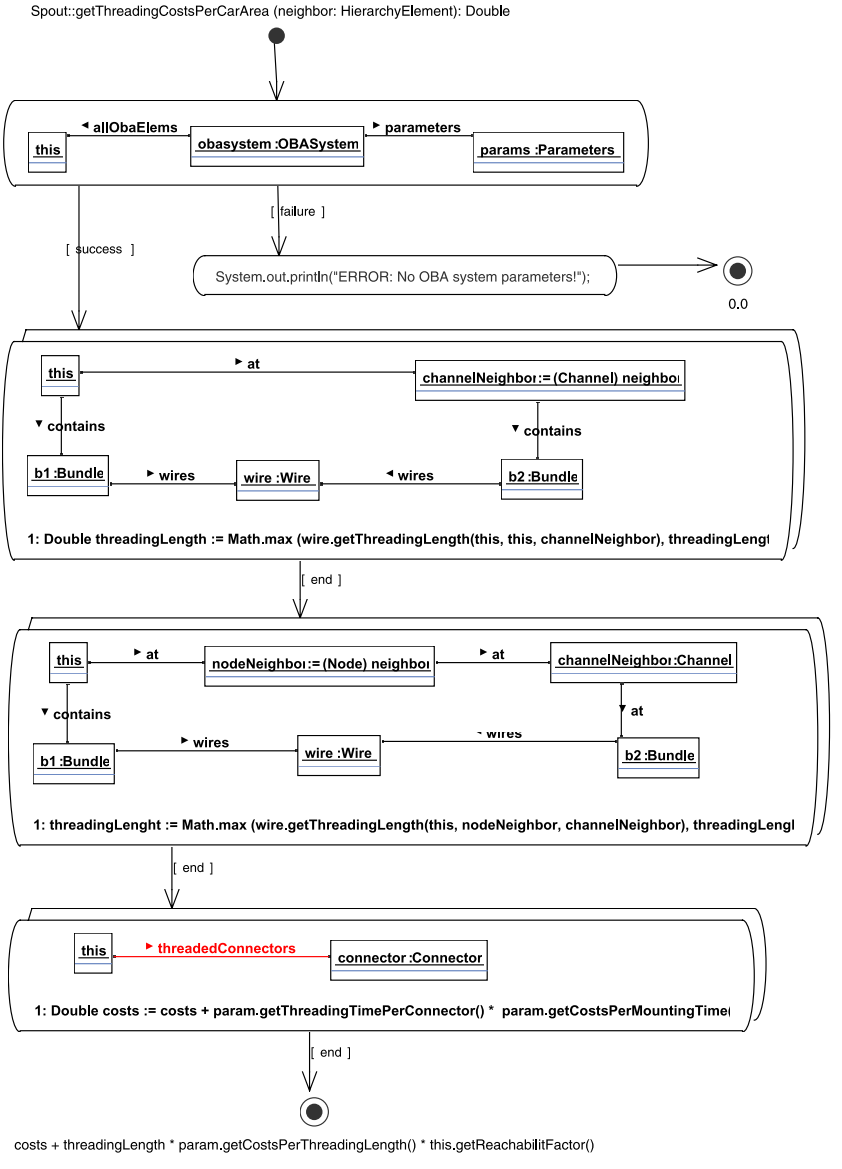


Fig. 8. Computation of the threading costs

to check whether these special cases were already addressed or whether the story diagram had to be adapted to handle such a special case, appropriately. Actually, in the threading length example, such a review discussion revealed that the first implementation attempt erroneously considered wires that only bypass a spout. Thus we had to extend the last but one story pattern of Figure 8 by the bundle object contained within the spout in order to ensure, that only threaded wires are considered.

To summarize the implementation phase, the use of story diagrams enabled our domain experts to review our implementation and to point us to special cases. In our experience, similar reviews by our domain experts would not have been possible for an implementation in a textual programming language like Java or C++.

4.4 Deployment

The Fujaba process focuses on the central software development activities. In the OBA project, it turned out that this is only a part of the overall project. For the deployment of the developed software, we needed not only to model the required car data but we had also to collect all the actual data of a car electrical system. The data on a car electrical system is distributed over a large number of heterogenous data sources. First of all, the car body geometry is provided by a CAD system owned by the OEM. In addition, the OEMs electrical engineers use another CAD tool to develop the circuit diagrams for the electronic system of the new car. In the next step the circuit diagrams are enriched with information on wire kinds, etc. This information is then joined with the car body geometry in order to determine the actual layout and length of cables. This is again enriched with information on required mounting and protection elements. As an example, Figure 9 shows a cutout of the final 3D CAD drawing for the wiring harness in the front left area of an example car. For the manufacturing of the wiring harness, the 3D CAD drawing is finally exported into a so-called *form board diagram*, a 2D flattening of the wiring harness.

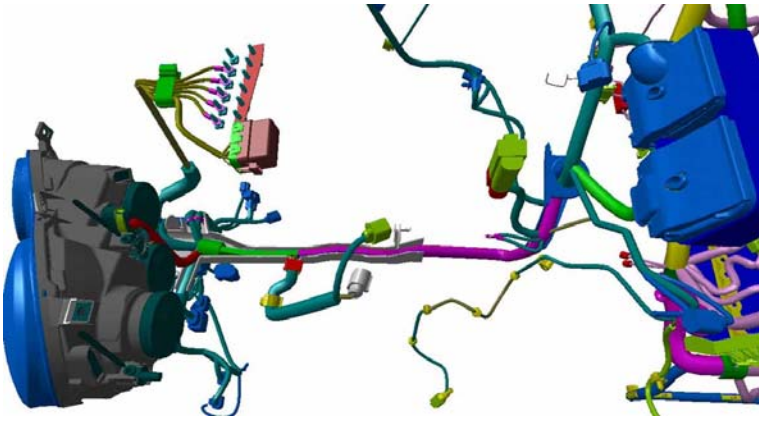


Fig. 9. Cutout of a 3D drawing of a wiring harness

While the described tool chain was well established at the industrial partners of the OBA project, the development process still involved many manual steps and frequently the data produced by one tool is interpreted by a human executing a subsequent step. Thus, we had to collect data e.g. from paper print-outs

and from several spreadsheets. While semi-formal, the spreadsheet data contains numerous data cells where automatic data import failed, e.g. because an engineer had entered some additional comment to the data. This is helpful for a human interpreting the data but crashes automatic data import routines. In addition, the data was frequently not on the level of detail that we needed for OBA. For example the size of connectors was usually not contained in the plans while OBA needed to ensure that the connectors still fit through hatches and do not exceed the size of e.g. the ECU they are connected to.

At the beginning of the process the domain experts where quite confident about the quality of the available car data. They have a well established tool chain and process and they are producing quite some cars with this process. However, with the help of the story board scenarios, it was easy for the team to elaborate which detail of data was actually required for the OBA tool. It was also easy to exemplify why e.g. certain comments in a spreadsheet cell are not easy to interpret for the OBA tool and that certain data like connector sizes are not yet available at all. This common understanding on data requirements was the key to solve the data retrieval problem. The common understanding was important because finally, the domain experts had to figure out where in the enterprises and in the existing tool chain such data might be retrieved. And, sometimes, the domain experts had to convince their colleagues that they have to extend the existing tool chain to provide additional data. This would not have been possible for the software developers from outside the enterprises.

5 Other Approaches

The Fujaba diagrams are derived from UML diagrams and try to stay as close as possible to UML. However, in UML related processes class diagrams have a much more central role and object diagrams are somewhat marginal. In the Fujaba process, object diagrams and story boards are central. Accordingly, Fujaba extends these diagrams, slightly. In SYSML, inner block diagrams are a big step in the direction of the Fujaba approach. These inner block diagrams allow to specify block instances and their relations at runtime and may be compared to our use of object diagrams. The main difference is that in SYSML, the blocks usually represent functional units that pass data values to each other via their connections. Restructuring operations on such nets of blocks are not central to SYSML. Thus, SYSML is well suited to model nets of functional blocks and how data is processed within such a net. Fujaba story boards are well suited to represent complex internal states of object oriented systems and how such complex states evolve during operation execution.

6 Lessons Learned

The main complexity of the OBA project was the sheer amount of domain details that contribute to the construction of a car electric system. The OBA project was initiated by our professor for car systems at Kassel University. Thus, the

project employed two domain experts from the research group of car systems. For the design and implementation work, there were only two PhD students from the software engineering group of University Kassel. At the beginning, the software engineering group thought that one domain expert and three software developers would make a better team. However, after roughly 6 years of software development, we have to state that the emphasis on domain experts was actually a key to success for the project. More precisely, since we had only two software developers, we were forced to involve the domain experts into the development process as much as possible. Using object diagrams we were able to do this in the requirements analysis phase. After a short learning curve, the domain experts were actually driving the requirements process. For each new aspect, they developed object diagram examples. These were discussed with the software experts in some iterations. Frequently, the domain experts from University of Kassel had to interview the domain experts in the contributing enterprises in order to clarify details. Thus, the domain experts at Kassel University served as some kind of mediators between the actual customers and the software developers. Through object diagrams, this worked out very well. Similarly, the contribution of detailed implementation reviews by our domain experts was of unmeasurable value for the project. Again, this was enabled by the object diagram like notation used in story diagrams.

For the deployment, the gathering of actual car data became a crucial issue. Many data came from tools without appropriate data export features. Some data was provided in paper format, only. And some data needed to be retrieved from the domain knowledge of the participating enterprises. Before the project, this data was used by human experts, only. While these experts were able to deal with slight inconsistencies and some missing data, the OBA tool requires consistent and complete data. Thus, the retrieval of the actual car data required a lot of negotiation with the domain experts working in the participating enterprises. During these negotiations it was crucial to communicate the detailed data needs of the OBA system and why the existing data sources failed to deliver the required details. Again, object diagrams turned out to be very helpful in communicating how OBA processes car data and which detailed information is needed for which calculation step. This helped tremendously to raise the mutual understanding of all partners and to ease the collaboration.

One may argue, that the Fujaba process requires the elaboration of a large number of object diagrams and story boards compared to a much smaller number of class diagrams used in conventional approaches. This is true. The cost model of the OBA project produced about 165 object diagrams and 107 pages of textual requirements. This resulted in 108 classes with 312 major methods which corresponds to about 8 class diagrams. However, creating the object diagrams is much easier than creating class diagrams, directly. In addition, skipping the object diagrams and creating the class diagrams directly would not have helped us to involve the domain experts. Many problems in our design would not have been discovered by the domain experts and we probably would have build the wrong system. In addition, after a short learning period, most of the object

diagrams have been created and elaborated by the domain experts. Thus, the domain experts did this extra work. Thereby the domain experts facilitated the work of the software engineers considerably. Actually, in other projects we made the observation, that software developers tend to need a much smaller number of less detailed object diagrams and story boards to cover some functionality. Non IT domain experts tend to use more examples and to elaborate them in larger details in order to ensure themselves that they did not miss any problem. In combination, the software developers do not need to invest too much time into the elaboration of object diagrams and the domain experts ensure that no important domain detail is missed.

Still, the large number of object diagrams create a certain maintenance problem. If the organization of the object structure changes reasonably, this may require only a small change to the corresponding class diagrams while this may cause changes in many object diagrams that contain the changed structure. For example, moving the temperature attributes from cable channels to car areas was a small change in the OBA class diagrams but quite a number of object diagrams needed to be adapted. Luckily, in the OBA project, again the domain experts did this work. Generally, improved editing support as e.g. copy and paste mechanisms and macro recording mechanisms need to be provided to minimize this effort.

Overall the incorporation of the domain experts worked out very well in this project. The key to success was, to enable the experts to contribute detailed domain knowledge within requirements analysis and even within the implementation phase. This worked due to the usage of object diagrams and story diagrams, i.e. due to graph grammar techniques.

References

[BRJ99] Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley, Reading (1999)

[DGZ05] Diethelm, I., Geiger, L., Zündorf, A.: Applying Story Driven Modeling to the Paderborn Shuttle System Case Study. In: Leue, S., Systä, T.J. (eds.) Scenarios: Models, Transformations and Tools. LNCS, vol. 3466, pp. 109–133. Springer, Heidelberg (2005)

[EMF] The Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf/`

[FNTZ99] Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Grammar Language based in the Unified Modeling Language. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)

[Fu09] Fujaba Homepage, Universität Paderborn, `http://www.fujaba.de/`

[GZ05] Geiger, L., Zündorf, A.: Story Driven Testing. In: Proc. 4th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM 2005), ICSE 2005, Workshop (2005)

[GSZ et al. 05] Gemmerich, R., Semmelrodt, S., Zündorf, A., Reckord, C., Leohold, J., Trippler, J., Brabetz, L., Müller, D., Schrey, U., Weil, H.-G.: An integrated approch for the generation and optimization of car electric systems. In: VDI (Hrsg.): 12th International Conference and Exhibition Electronic Systems for Vehicles Baden-Baden, pp. 597–608 (2005)

[JBR99] Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison Wesley, Reading (1999)

[KNNZ00] Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems. In: Proc. of ICSE 2000, Limerick, Ireland, pp. 241–251. ACM Press, New York (2000)

[UR05] Ungerer, M., Rabe, O.: VDA KBL - Harness Description List (KBL); VDA working group CAD/CAM (2005), http://www.ecad--if.de/documents/KBL_Data_model.pdf

[ZLM et al. 06] Zündorf, A., Leohold, J., Müller, D., Gemmerich, R., Reckord, C., Schneider, C., Semmelrodt, S.: Using object scenarios for requirements analysis - an experience report. Modellierung, Insbruck (2006)

[Zün02] Zündorf, A.: Rigorous Object Oriented Software Development with Fujaba (2002), http://www.se.eecs.uni--kassel.de/se/fileadmin/se/publications/Zuen02.pdf

Model-Driven Development with MECHATRONIC UML

Wilhelm Schäfer[1] and Heike Wehrheim[2]

[1] Heinz Nixdorf Institute,
Software Engineering Group
[2] Specification and Modelling of Software Systems,
Department of Computer Science, University of Paderborn,
Warburger Str. 100, 33098 Paderborn, Germany
{wilhelm,wehrheim}@upb.de

Abstract. Today, mechanical engineering products can largely be classified as *mechatronic systems*, i.e. systems integrating electrical and mechanical components with *software*. Mechatronic systems are often employed in safety-critical areas, for instance in the automotive or railway domain. MECHATRONIC UML is a UML profile specifically tailored towards the modelling of mechatronic systems. It aims at bringing model-based design and formal analysis to the mechatronic area, which has originally been dominated by engineering techniques. In this paper we give a survey of the modelling as well as verification techniques supported by MECHATRONIC UML.

1 Introduction

Today, software and its design is of critical importance for the proper functioning of so-called embedded or mechatronic systems. Mechatronic systems are characterised by a combination of basic mechanical devices with electrical and software components. Typical examples of mechatronic systems can be found in automotive applications, e.g. advanced braking systems, fly/steer-by-wire or active suspension techniques, but also DVD-players or washing machines. The control of these systems is handled by *software*. Sensors provide system and environment data to the control part, actuators allow to actively respond to such inputs.

In addition, advanced mechatronic systems [42] are today often characterised by a (very) high degree of distribution, i.e. they often consist of a large number of single entities which interact with each other. This interaction allows for a significantly extended functionality by exchanging information between system components which do not have to rely on sensor inputs only anymore. This can for instance be seen in the automotive and rail domain: Intelligent lighting systems combine information about their environment obtained from their own sensors with those collected by other cars. In the Paderborn rail system (introduced in more detail in the next section) shuttles autonomously form convoys as to reduce air resistance and optimise energy consumption. As a consequence, such systems often exhibit so called self-* properties, i.e. they adapt their own behaviour in reaction to the environment [29,30]. The enhanced facilities of gathering information both from sensors and from other entities create new possibilities for adaptation. Self-adaptation might concern changes of the particular component composition as well as of individual component behaviour.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 533–554, 2010.

Besides self-adaption, there is another characteristics of advanced mechatronic systems which makes their design hard. Communication between system components means the exchange of complex state information which influences the control of system components significantly. The interplay between the continuous control based on sensor input on the one side and the discrete control based on communication on the other side leads to usually very complex software. In general, mechatronic systems fall into the category of *hybrid systems* involving continuous as well as discrete parts. The design of mechatronic systems thus means designing a hybrid, self-adapting system.

Mechatronic systems are often employed in safety-critical contexts. To guarantee high quality of these systems in practice, usually large numbers of tests based on simulation are being run. The (software) complexity of safety-critical systems however requires more sophisticated methods than "only" simulation and testing; techniques which give us a high guarantee, at the best a proof of correctness. In traditional software development, today this ambitious goal is approached by a model-based design which is complemented with a *formal analysis*. The main objective of MECHATRONIC UML is to bring this approach to the world of mechatronic systems. Our research focuses on complementing testing and simulation with model based design and formal analysis to guarantee highly safety-critical system properties.

To this end, we have taken UML as the standard for software modelling as starting point. MECHATRONIC UML is a specific profile for mechatronic systems, which on the one hand restricts the use of UML to certain diagram types and on the other hand specialises these diagrams for the usage of modelling hybrid, self-adapting systems. For modelling self-adaptation on the component structure level we employ graph-transformation systems. To allow for a formal analysis, all modelling elements in MECHATRONIC UML furthermore have a formally defined semantics.

In addition, our approach is based on a formal architecture definition. The proper definition of component interfaces enables a verification of inter-component communication based on formally specified protocols which are attached to specific ports. The interface definition guarantees that no such protocol influences the behaviour of another one, i.e. protocols are specified such that no side-effects occur. Scalability of our approach to real systems thus becomes possible by applying compositional verification. Other special architectural modeling approaches like SysML [34] and the UML SPT [37], or rather its successor MARTE [36], do not adequately support modeling of time (SysML) or do not provide the needed architectural abstraction from hardware details (MARTE).

Our architecture definition also supports the tight integration between the definition of controllers by block diagrams and differential equations and the definition of state-based behaviour by timed state charts. In contrast to others [1, 24], we abstract from the detailed definition of controllers by a proper interface definition which enables to verify the timed state charts independently from the controller input/output behaviour thus improving scalability. The correct behaviour of controllers as well as the time needed for reconfiguration of controllers is assumed to be guaranteed by control engineers and their corresponding tools. The MECHATRONIC UML approach thus supports a tight interaction of the disciplines involved in designing a mechatronic system, the mechanical and electrical engineers and the software designers.

In summary, model driven development using MECHATRONIC UML approach includes the following steps: definition of the system architecture by an (extended) UML 2.0 component model, specification and verification of individual component behaviour by a special extension of timed state charts to address hybrid systems, specification of system reconfiguration, i.e. architectural changes to address self-* properties, by timed graph transformation systems and finally fully automatic code generation.

The paper is structured as follows. In the next section we introduce our main application, the Railcab project. This serves as our running example for modelling and analysis with MECHATRONIC UML. Section 3 will introduce our general architectural model. Sections 4 to 6 describe modelling and verification with MECHATRONIC UML. Section 7 gives a short introduction into our tool FUJABA supporting both modelling and verification. The last section concludes and presents future work.

2 The Railcab Project

The Paderborn-based RailCab research project (http://www-nbp.upb.de/en) is a concrete example for a mechatronic, hybrid system exhibiting self-adaptation. It aims at combining a passive track system with intelligent shuttles that operate individually and make independent and decentralized operational decisions. The project is funded by a number of German research organizations. It has built a test track in the scale of 1:2.5 such that the ideas of the project are not only tested "on paper" but in real operation.

Fig. 1. Railcabs driving in a convoy

The vision of the RailCab project is to provide the comfort of individual, traffic concerning scheduling and on-demand availability of transportation as well as individually equipped cars on the one hand, and the cost and resource effectiveness of public transport on the other hand. The modular railway system combines sophisticated undercarriages with the advantages of new actuation techniques as employed in the Transrapid

(http://www.transrapid.de/en) to increase passenger comfort while still enabling high speed transportation and (re)using the existing railway tracks.

One particular aspect is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they build convoys whenever possible. Such convoys are built on-demand and shuttles travel only a few centimeters apart from each other (up to 0.5 m) such that a high reduction of energy consumption is achieved. This requires a lot of information exchange between the various machines or system components like the shuttles, registrars, dispatchers, stations, customers, etc. It also means a tight integration of quasi-continuous and discrete control software and the realization of complex functionality by software rather than hardware. For example, travelling only at a few centimetres distance in a convoy requires tight coordination between the various speed control units under hard real-time constraints.

In the following, we will use the Railcab project and, in particular, the convoy building of shuttles as our running example for MECHATRONIC UML modelling and analysis. It exhibits the main characteristics of advanced mechatronic systems and the challenges that this imposes on software design: the need for integrating continous control with discrete software and the need for self-adaptation.

3 Architectural Model

Our approach to the model-based design of mechatronic systems consists of several phases, which incorporate different modelling as well as verification activities. All of these phases require a tight interaction of software and control engineers as both sides are involved in the decisions being made. The first phase consists of specifying the overall architectural structure of the system. This will first of all be given as a hierarchy of *Operator-Controller-Modules* (OCMs) (1), as described below. Based on this model, a more detailed architecture modelling, in particular of component *interaction*, is specified via UML 2.0 component diagrams (2). The next phase consists of the detailed design and analysis of the interactions themselves. To this end, all connectors and associated ports in the component diagram are attached to a so-called *coordination pattern* (3). Coordination patterns describe interaction protocols, possibly including timing constraints. Via model-checking we analyse the correctness of the composition of such interaction protocols (4). The next step consists of the modelling of components themselves: we specify the behaviour of components with a form of hybrid statecharts (5), in which the usage of particular controllers is attached to states. The reconfiguration of controllers is build into these statecharts, and includes the definition of so-called *cross-fading functions* to allow for a smooth switching of controller. Such a modelling of components requires two more correctness checks. First of all, we have to show that the component behaviour is consistent with the specified interaction protocol (6), and second we need to show that the timing conditions in the statecharts are consistent with the physical world, i.e. for instance actually represent the fading duration of a cross-fading function (7). The latter aspect is a task of the engineers. Finally, going back to the top level of the architecture, one more modelling possibility is reconfiguration on the architectural level (specified by graph transformation systems), which gives us the flexibility of adapting in particular software protocols to changing environmental situations (8).

This section will describe the modelling of the structural level only, (1) and (2), the next sections have a closer look at component interaction and controller integration.

The Operator-Controller-Module, as depicted for our example in Fig. 2 (cf. [27]), describes a general architectural model of a single system component and identifies its constituent parts.

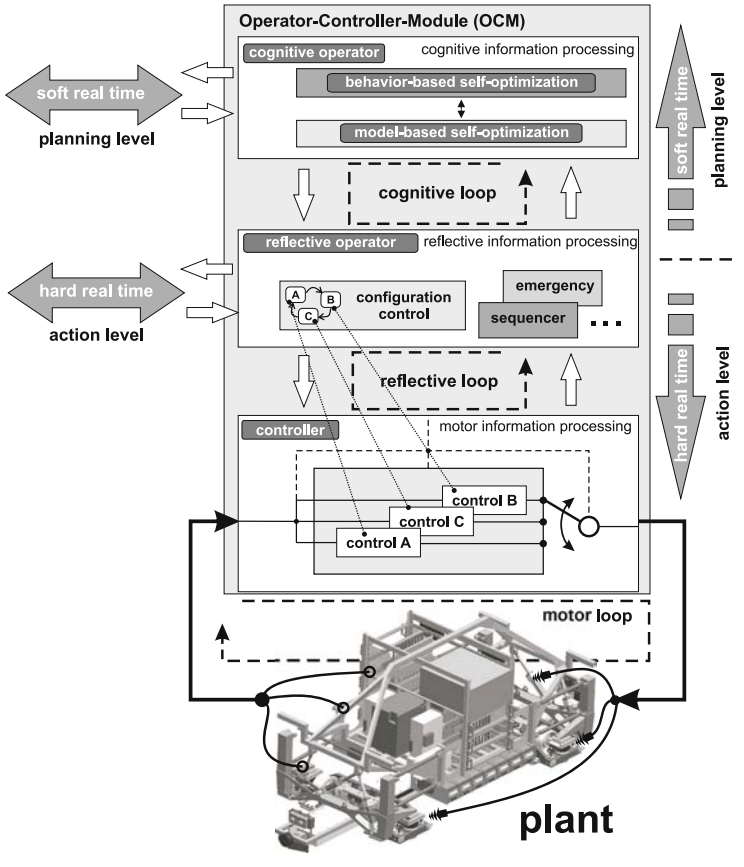


Fig. 2. OCM architecture

The OCM reflects the strict hierarchical construction of a system component including the hardware components: (1) On the lowest level of the OCM, a more or less standard controller (C) implements a feedback loop based on sensor input and producing actuator control signals accordingly. This is called the motor loop. The software processing is necessarily quasi-continuous, including smooth switching between possible alternative control strategies which are described by some form of differential equations or difference equations. (2) The controller is controlled by the reflective operator (RO), in which monitoring and controlling routines are executed. The reflective operator operates in a predominantly event-oriented manner and thus includes a control automaton

with a number of discrete control states and transitions between them. It does not access the actuators of the system directly, but may modify the controller and initiate the switch between different control strategies. Furthermore it serves as the connecting element to the cognitive level of the OCM. (3) The topmost level of the OCM is called the cognitive operator (CO). On this level, the system can gather information concerning itself and its environment and use it for the improvement of its own behaviour, i.e. possibly complex, time-consuming computations for long-range planning. This level introduces intelligent behaviour and consequently such components could also be called agents.

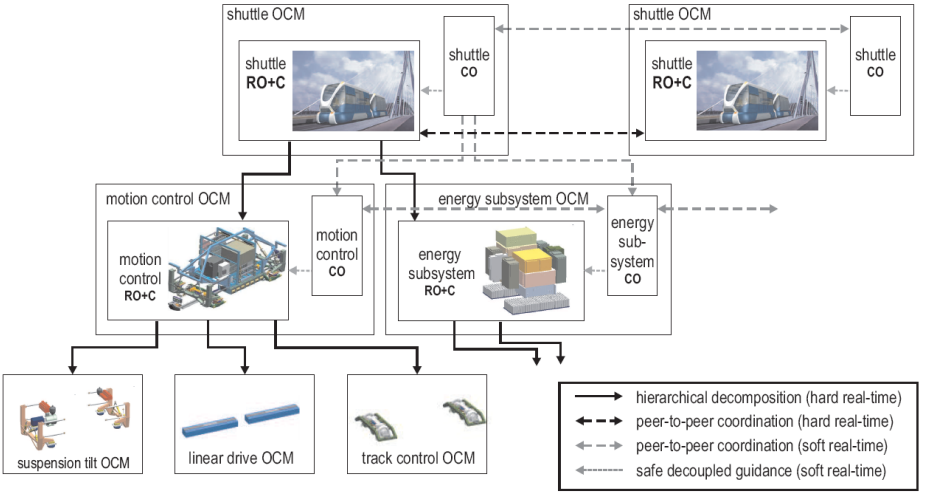


Fig. 3. The OCM hierarchy of a shuttle and its connections with other shuttles

The OCM hierarchy can be nested, i.e. each nesting level may include an OCM which however does not include the controller part. Controllers, which implement the continuous part of the software, usually exist only on the leaf level of a nested OCM hierarchy. As an example, consider the above mentioned shuttles of the RailCab project. The architecture is defined by OCMs w.r.t. the reflective operators and the controllers as depicted in Fig. 3. A shuttle consists of components like the suspension/tilt module, the engine, or the steering module. They will utilize the same hardware (actuators, sensors and controller) but their software is defined by OCMs as is the software of the shuttle.

As a complete mechatronic system usually consists of several concurrently running components, there exists furthermore the possibility of *communication* between components besides the strict hierarchical control flow within one component. OCMs may also act as freely interacting system components (see Fig. 3) where controllers, reflective and cognitive operators resp. of one component exchange information with their counterparts of another component. As examples consider communication between system components like the different shuttles, stations, job brokers and dispatchers of the RailCab project.

Based on this general model, a (slightly extended) UML 2.0 component model is used to represent the software architecture of a concrete system in detail. A key feature of this architecture is the detailed definition of the individual component ports and their connections with other ports. This concerns the component hierarchy of a single component which corresponds to a single OCM as well as the peer-to-peer communication between top-level OCMs. We also distinguish between ports which represent the exchange of state information on the level of reflective and cognitive operators, and ports which represent the exchange of controller information based on sensor input. The latter, called *continuous ports*, are displayed as squares with an arrow inside, the former ones are simply squares and thus follow the standard UML notation.

Fig. 4 illustrates the UML 2.0 component model representation of the OCM of a single shuttle. The (communication) links (connectors) define port to port communication, i.e. they identify all possible intra- and intercomponent interactions. Each of these links corresponds to a communication protocol as explained in the next section. The inputs and outputs via continuous ports are defined by a controller definition in terms of block diagrams and corresponding differential equations.

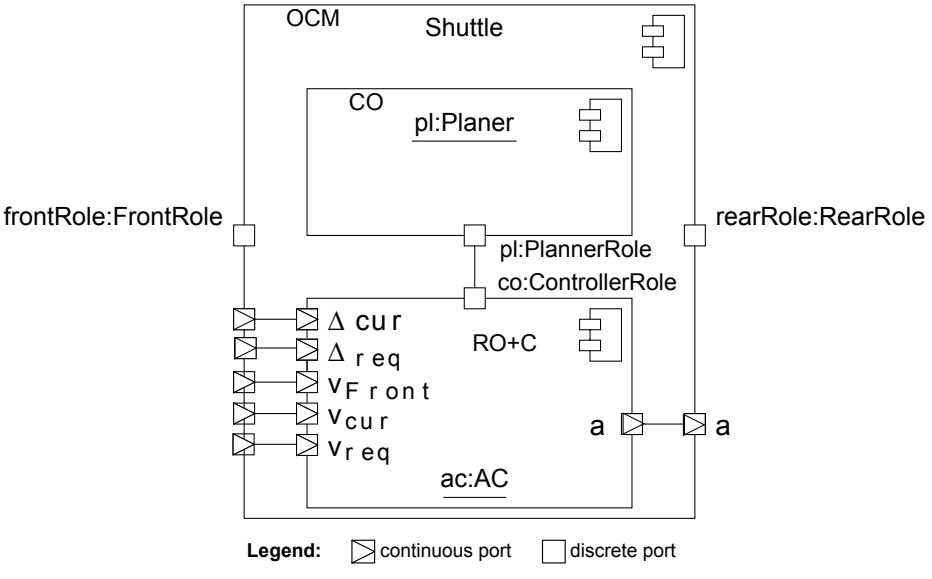


Fig. 4. Example for a component structure of a Shuttle OCM

4 Protocol Modeling

Having specified the component structure, we next develop a more detailed description of the communication links. Each communication link is specified by a so called (real-time) *coordination pattern*. These patterns are a refinement of the loosely defined collaboration and pattern concepts in UML 2.0. Coordination patterns mainly fix communication protocols. As we assume a port to port communication based on one such

pattern, no interference with other patterns (or rather, their instances in a concrete system) can appear. This allows us to individually verify patterns, and afterwards compose verification results (compositional verification).

A coordination pattern consists of a set of roles that communicate only via ports based on a related connector that connects those ports. Each role and connector in turn are specified by a real-time state chart (RTCS). RTSCs are an extension of UML state charts especially supporting the notion of *time*. This allows for modelling timing constraints for transitions and states like e.g. deadlines, lower and upper bounds or worst case execution times (WCET). To support model checking of safety and lifeness properties of the communication protocols specified by RTSCs, they are mapped to timed automata which additionally gives them a formal semantics. (On a side note, this means abandoning the run-to-completion semantics of UML state charts which is not appropriate for modeling real-time and mechantronic systems.) In fact, RTSCs are an abstraction of timed automata, that support building more comprehensible models as they are usually smaller than the corresponding timed automata. For details we refer to [11]. As a model checker for the generated timed automata, we employ UPPAAL [6]. We use it basically for two kinds of analyses: on the one hand, we show that the controllers (see next section) associated with states in the state charts are correctly embedded. On the other hand, we can prove time-dependent safety properties of our systems on the generated timed automata.

The specification of safety properties on communication protocols is given by declarative constraints which are defined using temporal logic. As nonfunctional system properties such as real-time behaviour are not supported by the standard UML and related extensions for real-time systems such as the *UML Profile for Schedulability, Performance, and Time* [37], a state-based temporal extension of the Object Constraint Language (OCL) called RT-OCL [16] is used. As the examples in this paper only contain formulas in pure OCL, we further omit any details of RT-OCL here.

The communication between shuttles which is necessary to build a convoy (of, in this case, two shuttles) is one such coordination pattern. Fig. 5 shows a ConvoyCoordination pattern instance between two shuttles. It defines a drastically simplified protocol for building and breaking convoys based on two roles, namely the rear role and the front role (see Figure 5). Initially, both roles are in state noConvoy::default, which means that they specify the situation where a shuttle is not a member of a convoy. The rear role decides whether to propose building a convoy or not. After the decision has been taken, a message is sent to the other shuttle, or rather its front role instantiation. The front role decides to reject or to accept the proposal after max. 1000 msec. In the first case, both statecharts revert to the noConvoy::default state. In the second case, both roles switch to the convoy::default state.

Eventually, the rear shuttle decides to propose a break of the convoy and sends this proposal to the front shuttle. The front shuttle decides to reject or accept that proposal. In the first case, both shuttles remain in convoy-mode. In the second case, the front shuttle replies by a confirmation message, and both roles change to the corresponding noConvoy::default states.

The connector which represents the wireless network does not need to be specified by an explicit statechart specification here, but instead by its QoS characteristics such

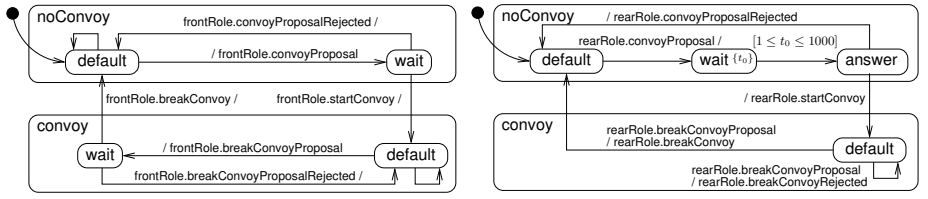


Fig. 5. Statechart of the rear role (see Figure 5(a)) and the front role (see Figure 5(b))

as throughput, maximal delay etc. in the form of *connector attributes*. In our example, we assume that the connector forwards incoming signals with a delay of 1 up to 5 msec. The connector is unsafe in the sense that it might fail at any time, such that we set our specific QoS characteristic reliable to false.

A safety property of this pattern is CanBrakeFully which describes that a shuttle can only do an emergency brake when it is not taking the front position in a convoy. In detail, the property specifies for any rear role instance that being in state convoy implies that CanBrakeFully holds. In contrast, for any front role instance, state convoy requires that CanBrakeFully does not hold. The following OCL role invariants are used to describe these restrictions.[1]

```
context <component> inv:
  <frontRole>.oclInState(convoy) implies          (1)
    not self.CanBrakeFully

context <component> inv:
  <rearRole>.oclInState(convoy) implies           (2)
    self.CanBrakeFully
```

As the specified state chart model includes the notion of time, time-critical safety properties are also verifiable. As an example, consider that negotiation about forming a convoy between a number of shuttles (all of them approaching a switch) may not exceed a given threshold. This guarantees that emergency actions might still be possible in case, for example, a communication link is broken.

When defining the behaviour of a component like a shuttle using these patterns, the predefined role behaviour has to be refined and synchronized. The following example illustrates this step. Fig. 6 depicts the behaviour of the Shuttle component from Fig. 3. The RTSC consists of three orthogonal states FrontRole, RearRole, and Synchronization. FrontRole and RearRole describe the port behaviour. They are refinements of the role behaviours from Fig. 5, and specify in detail the communication that is required to build and to break convoys. An additional internal RTSC is used to specify the synchronization between these roles as well as the embedding of a hybrid component hierarchy. In our example, Synchronization coordinates the communication and is responsible for initiating and breaking convoys. The three sub-states of Synchronization

[1] The context component enclosed in angle brackets is employed here as a placeholder for the component which realize the role via one of its ports.

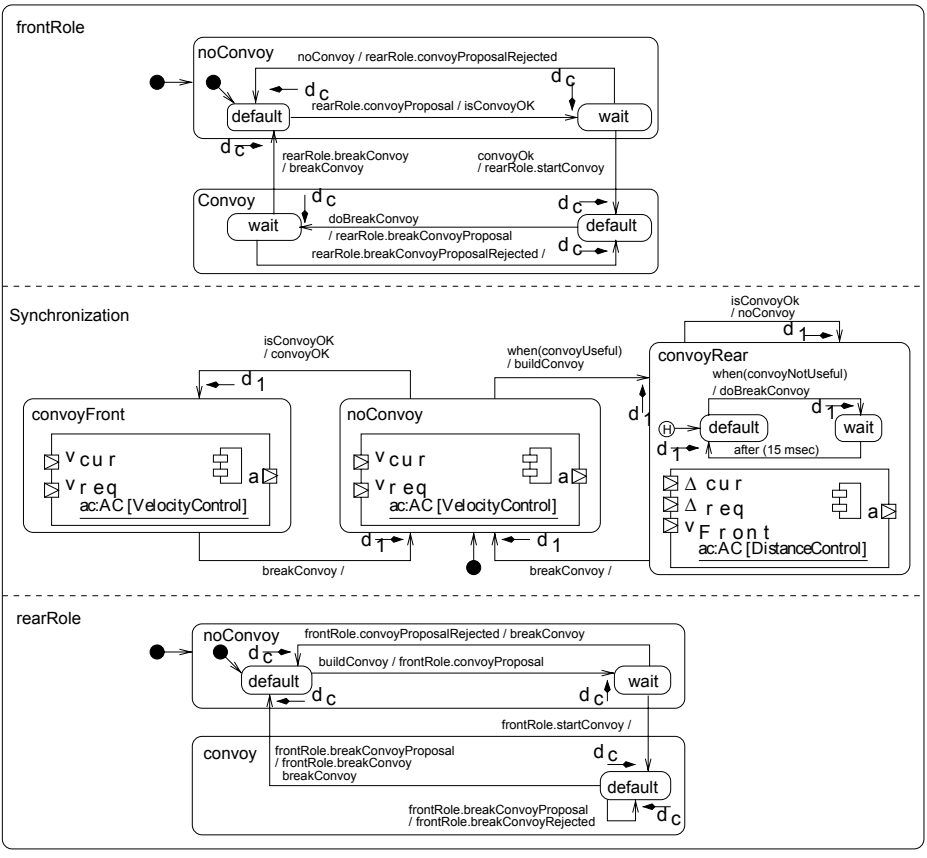


Fig. 6. Behaviour of the Shuttle component

represent whether the shuttle is at the first position (convoyFront), at second position (convoyRear), or no convoy is built at all (noConvoy). At this point, the reader should ignore the embedded states which correspond to controllers and will be explained in the next section.

Strictly defined refinement rules, which are checked syntactically, make sure that the refinement of role behaviour does not invalidate the verification results of model checking the coordination patterns. Basically, they only allow to refine the nondeterministic behaviour of the patterns to a deterministic one. Then the final verification step to be done is to verify that the newly constructed synchronization state chart does not include any deadlocks which is checked using UPPAAL again.

We have given a formal proof that compositional verification based on constructing a complete system just by the above sketched definition of components, ports and refined coordination patterns is indeed possible [19].

Admittedly, there is still one significant limit of the approach. Broadcasting based communication architectures cannot be covered by our approach. However, we found

many examples of systems where this type of generality does not exist and thus our approach is applicable to a wide range of systems like e.g. public transport, production or telecommunication systems. In fact, one could even argue that building "really" safe systems requires to avoid broadcasting-like communication structures.

5 Controller Reconfiguration

After specifying component behaviour by coordination patterns, we now address the connection of two different levels in the OCM hierarchy, namely the detailed definition of connecting the reflective operator and the underlying controller. Based on the component communication, the reflective operator initiates the use of a particular controller and, if necessary, the *reconfiguration of controllers*, i.e. the exchange of one or more controllers by one or more others.

Extending RTSC to specify the connection between discrete and continuous behaviour is done similarly to the basic hybrid automata approaches like [25,7,2,32] by the possibility to assign a configuration of controllers to a particular state for Hybrid RTSCs (HRTSCs).

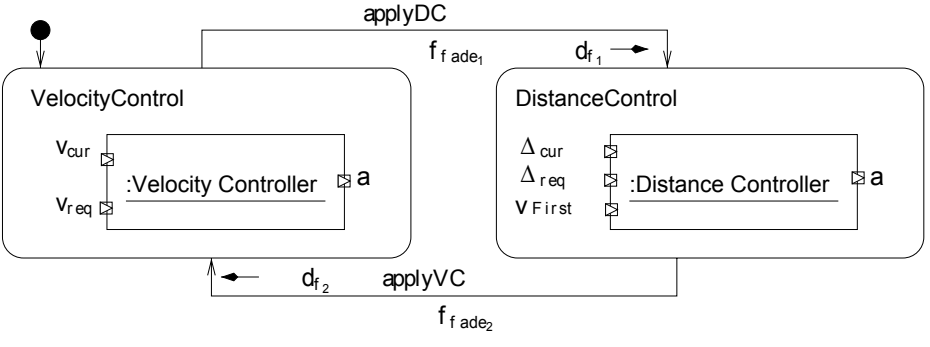


Fig. 7. Behavioural embedding

Take as an example the hybrid behaviour of an Acceleration Control (AC) component which is embedded into the Shuttle component. It consists of two discrete control modes which specify whether the shuttle is operating in velocity control mode or distance control mode respectively (see Fig. 7). Furthermore, it has continuous in- and outputs. Depending on the active discrete mode, the current and the required velocity are used as input, or the current and required distance to the front shuttle plus the velocity of the first shuttle are used. The output a is the acceleration in both modes.

This example does not illustrate a syntactic feature of our HRTSCs which is however worth mentioning to understand Fig. 6 in full detail. As the component model only defines all possible input and output ports, there remains a certain ambiguity on the controller level. The same controller may have different input or output ports depending on the particular usage. In the example above this is not the case but may well be for velocity control. The particular usage is depending on the system state. That is why we add the definition of input and output ports to the HRTSC definition as seen in Fig. 6.

Of course, consistency between the component model and the HRTSC specification concerning the names and connections of the ports is automatically checked.

In our example, each configuration consists of one single feedback controller while in the general case a configuration of subordinated blocks representing a number of (continuous) controllers might be assigned to each state. In addition to specifiying controllers, switching smoothly between different controllers requires to define output *cross-fading functions* (cf. [17]). Here, we have the fading functions f_{fade_1} and f_{fade_2}, and a minimal and a maximal *fading duration* (d_{f_1} respectively d_{f_2}) which together specify how the outputs of the two controllers have to be faded when changing the controller. The timing constraints given by the fading functions are mapped to corresponding *deadlines* in the HRTSC. In the example depicted in Fig. 7, the state-dependent continuous behaviour is specified by blocks assigned to the states VelocityControl and DistanceControl, respectively.

In cooperation with the control engineering department, we have also developed other possibilities to precisely specify timing constraints for switching between two controllers. For details we refer to [38]. To cover nested components with changing input/output (i.e. OCM hierarchies with reconfiguration), we have furthermore introduced an extension of the known concept of hybrid behaviour which assigns a configuration of embedded (possibly hybrid) component instances to each state instead of control behaviour only. For details, we refer to [17].

Model checking HRTSCs is currently based on a mapping to the Hybrid model checker PHAVer [23]. However, it is clear that this approach only works for small examples. We are working on improvements which are based on identifying appropriate intervals for the continous input/output relations based on application specific information rather than taking the predefined PHAVer discretisation.

Similar to the above described approach, the behaviour of the cognitive operator is also modelled by state charts. On this level, however, time is much less critical and thus does not need to be modelled and analysed. As said, this part of a system component is responsible for long range planning. Results of such a long range planning process are transferred via the corresponding port definitions to the reflective operator, i.e. they appear as activities of a HRTSC and may trigger architectural reconfiguration of a system (cf. Section 6). This type of reconfiguration, in contrast to the discussed reconfiguration of controllers, may mean adding or deleting complete system components or communication links. How to model check this type of reconfiguration is explained in the next section.

6 Architecture Reconfiguration

RTSCs and HRTSCs describe behaviour and self-adaptation of controllers on the level of single components. Advanced mechatronic systems, and, for example, the shuttles of our Railcab project, in addition employ reconfiguration on the level of the overall system structure, changing the relationships and interaction between components. This is part of the cognitive operator of the OCM. As an example, consider that shuttles in a convoy have to *consistently* decide for driving (or not driving) as a convoy. A positive decision involves changing the protocol of interaction between shuttle as convoy or noConvoy mode which means to employ different protocols. As this reconfiguration

involves usually quite a bit of restructuring the architecture of the system, i.e. deleting and changing objects and corresponding links to other objects, using state chart based models like in the previous section is not appropriate. State charts do not support adequately the modelling of complex graph transformations like architectural changes but rather "only" the replacement of a complete configuration by another one like the replacement of controllers. Thus, in this section we base both reconfiguration modelling as well as their formal analysis on Graph Transformation Systems (GTSs).

Architectural changes which have to be modelled, concern e.g. creating or deleting system components and corresponding communication links like shuttles but also changes of the behaviour of communication links which are specified by the described *coordination patterns*. A coordination pattern fixes a certain protocol of interaction (as specified by an RTSC) and some particular controller usage in the mechanical parts (fixed in the HRTSC). Coordination patterns (and thus the protocols and controllers) can be activated or deactivated depending on the current environmental situation of a component. More sophisticated systems might even load and unload the software for pattern protocols by demand during run-time, i.e. software is physically stored or is not stored on a particular device like a shuttle. In the embedded area this is often necessary as resources (e.g. storage) are severely limited. Examples of communication changes are shuttles that only need to communicate with a registry when they are in the registries surveyed track section (every track section is monitored by a registry, and shuttles need to register when they enter a registry's section), or shuttles which only have to communicate with other shuttles when they intend to form a convoy.

This reconfiguration (activation and deactivation of patterns), but also the structural changes due to "normal" system evolutions, are described by graph transformation *rules* (or, more generally, *story patterns*, additionally allowing to specify orderings on reconfigurations). In our approach, a system state is characterised as a configuration of components (e.g. a shuttle, a track), their relationships (e.g. whether a shuttle is standing on a track) and their currently used coordination patterns. Figure 8 exemplifies one such

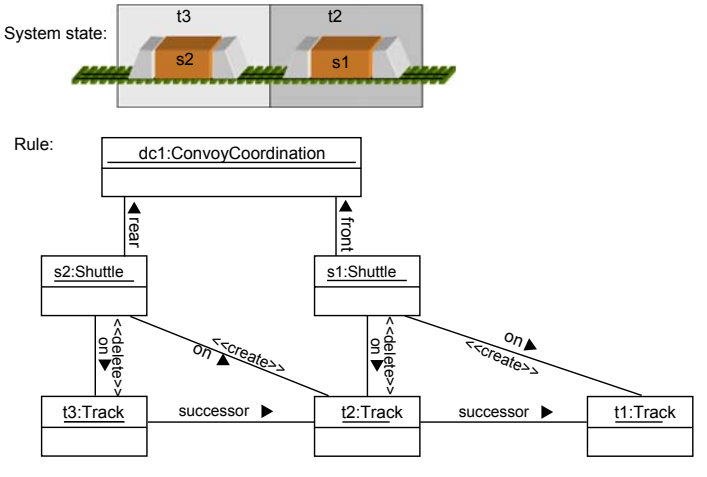


Fig. 8. Reconfiguration as graph transformations

scenario. The upper part of the figure shows a concrete system state where two shuttles are driving on two different but neighboured tracks. The lower part of the figure shows the coordinated movement of shuttles on neighboured tracks, however, only when both have instantiated the ConvoyCoordination pattern. The system configuration is given as a graph where the nodes are components or patterns and the edges are relationships between components. For instance, the shuttles take different roles when participating in patterns (front or rear role) and this is depicted as a correspondingly labelled edge between a shuttle and the pattern node. Reconfiguration, i.e. here the movement, is modelled by a *graph transformation rule*: the left hand side identifies the system configuration in which the rule can be applied, the right hand side shows the result of application. Here, left and right hand side are depicted in one single graph, using the notation $\ll create \gg$ and $\ll delete \gg$ to specify edges (and nodes) which are created viz. deleted during rule application. Similarly, for this example, we have rules which specify when the ConvoyCoordination pattern is allowed to be activated and deactivated. Altogether this forms a graph transformation system, consisting of a set of reconfiguration rules plus possibly an initial configuration given by a plain graph.

The semantics of graph transformation systems is well defined, thus we again have a formal model here. For the semantics we follow the SPO approach [15]. The semantics gives us the possibility of carrying out a formal analysis on the model. Safety properties which need to be guaranteed by the system, in particular under reconfiguration, are specified as *graph patterns*. Graph patterns define structural properties; properties of the underlying controllers (like stability) are not treated on this level, rather, need to be guaranteed by the control engineers. Graph patterns can either model forbidden or desired configurations.

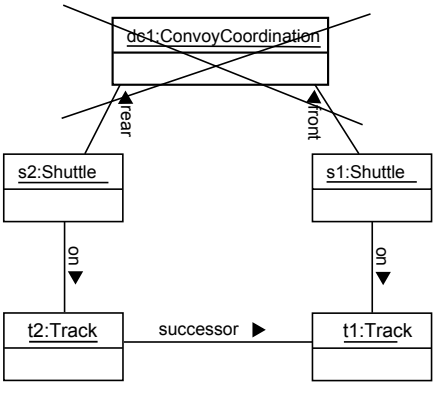


Fig. 9. A forbidden graph pattern

The example in Figure 9 shows a configuration, where two shuttles on neighboured tracks have not agreed on using the ConvoyCoordination pattern. This is potentially unsafe since the protocol for coordinated movement is then disabled. For verification, we

show that none of the reachable graphs exhibit such a forbidden graph pattern. Currently, there are two approaches available for proving these safety properties:

Model checking. This method follows the traditional technique of model checking building the complete set of reachable graphs given a start graph and a set of rules. As prerequisites we thus first of all need such a start graph, which is not neccessarily always given, plus we need to ensure that the set of reachable graphs is finite. If this is the case, the technique works as follows: the reconfiguration rules given as story patterns as well as the start graph are translated into the formalism used by the model checker Groove [40]. Groove is a verification tool for graph transformation systems. Groove subsequently builds the state space of the graph transformation system, and on this we can check our safety properties.

Induction. Induction [5] is applied when either the existence of a start graph or the finiteness condition cannot be guaranteed. The technique proceeds by inductively showing that the absense of certain graph patterns is fulfilled for all graphs created by the rules. To this end, the rules need to be transformed: instead of deleting edges, edges are specifically marked as deleted as to be able to track deletion. Induction is then carried out from backwards: we show that a forbidden graph pattern can only be reached by a rule application when the graph to which the rule is applied already contains this forbidden pattern. If this can be guaranteed for all rules, the absense of the forbidden pattern forms an inductive invariant of the GTS.

Both techniques have been implemented within the FUJABA tool suite (see next section) and show good performance on examples from our application domain.

What is not captured by these approaches, but is relevant for the domain of mechatronic systems, is the *time* needed for a reconfiguration. In particular, the activation or deactivation of coordination patterns which involves changes of communication protocols and controllers may need a significant amount of time. This timing aspect is of importance as the shuttles travel with a certain speed on the tracks, and safety properties might depend on fast reconfiguration. For capturing timing aspects in reconfiguration, graph transformation rules have been extended with time. Figure 10 shows an example of a timed rule. The timing condition describes the time units needed for carrying out the reconfiguration. Here, this is the minimal amount of time needed by a shuttle to move from one track to the next.

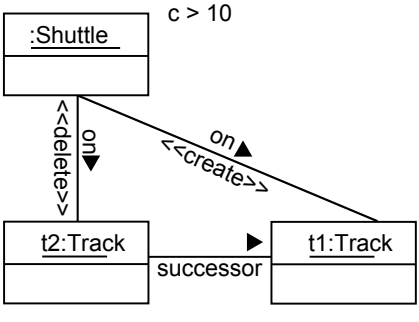


Fig. 10. A timed graph transformation rule

For timed graph transformation we have also developed and implemented a verification technique following the classical approach of state space generation. To this end, we have extended Groove with timing. Technically, we use clock zones to reduce the infinite number of timed configurations to a finite number. This follows an approach used in timed automata model checkers like UPPAAL.

Such an approach enables to verify time critical safety properties like "the change of all protocols in all shuttles interested in forming a convoy happens in a certain time frame, e.g. 5 milliseconds". This might be necessary to still be able to take emergency actions in case one of the communication links between the concerned shuttles may be broken.

7 The FUJABA Real-Time Tool Suite

The FUJABA real-time tool suite is part of the open source tool suite FUJABA (acronym for "From UML to Java and back again") which was kicked off by the software engineering group at the University of Paderborn in 1997. In 2002, FUJABA has been redesigned and became the FUJABA tool suite with a plug-in architecture allowing developers to add functionality easily while retaining full control over their contributions. Different FUJABA tool suites exist, which all include a common core functionality and a number of different plug-ins. The plug-ins provide support for different domains and different activities of the software life cycle. The FUJABA real-time tool suite, in particular, includes support for modelling and analysing real-time embedded systems, code generators for different target platforms and a tool enabling hazard analysis [39].

Tool support for modeling and checking system specifications, as given by the above described component models and HRTSCs, has been implemented as part of the FUJABA real-time tool suite. In order to support the specification of controllers by block diagrams and differential equations a commercially available tool, namely CAMeL-View, has been integrated into the FUJABA tool suite.

Both tools offer code generators which target various platforms. In particular, both tools provide a C++ code generator. FUJABA generates its code from HRTSCs, whereas CAMeL-View takes the block diagrams and differential equations as input [10].

A key aspect in integrating two tools from "different worlds" is the invocation of code which is generated from the block diagrams in CAMeL-View, by the code generated from the hybrid statecharts excluding the controller part. Controllers and in particular their feedback loops run concurrently with the execution of statecharts or rather the generated code. Particular synchronization points have to be identified in the controller code, where state changes and consequently controller reconfigurations are possible. For more details see [9].

As a simulation platform called IPANEMA [12] exists which executes C++ code, the resulting code is used to simulate complete system behaviour. In additon to the three dimensional CAMeL-View simulation of controller behaviour, we are now able to also display the corresponding state information of each system component as well as their current communication partners.

In contrast to most other approaches, a platform independent model specified in MECHATRONIC UML has enough information to generate code fully automatically.

This means in particular that the specification by RTSCs contains all real-time constraints which a system has to fulfil and consequently have to be guaranteed by the implementation. [13] describes an algorithm which derives a single processor implementation in Java from a MECHATRONIC UML model such that all real time constraints are either fulfilled by the implementation or the algorithm signals the user that such an implementation is impossible. The algorithm is based on a schedulability analysis based on the processor cycle time and on swapping the code of actions as given by the state chart specification.

Fig. 11 shows a screenshot of the currently available tool. It is an extension of the CAMeL-View simulation environment. The three dimensional view in the lower part simulates a convoy of two shuttles. State information can now be displayed in addition to the controller status information of all shuttles (lower right part). The graphs in the upper part of the screen display "traditional" controller information. In detail, the upper right part displays the position of the shuttles, the upper left hand the velocity of the first and second shuttle, and in the lower left the reference values for speed are shown.

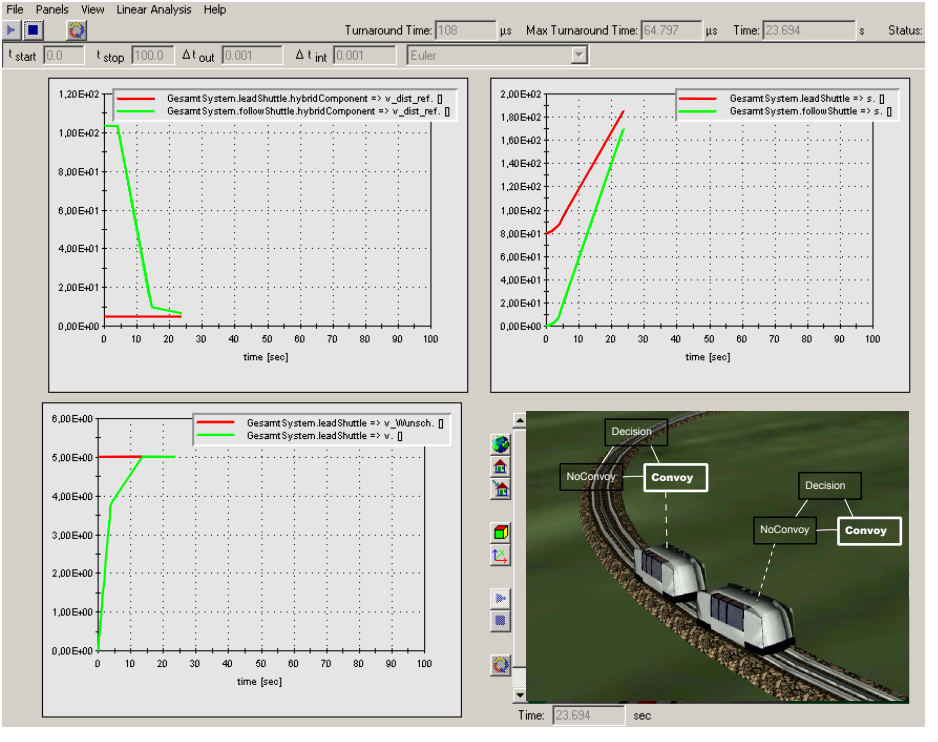


Fig. 11. Simulation Environment

The sketched tool, or rather its developers in the Software Engineering Group at the University of Paderborn, were recipients of the IBM Real-time Innovation Award 2008 for this achievement.

8 Related Work

The UML itself and even UML extensions for real-time such as the UML Profile for Schedulability, Performance, and Time [33] are not sufficient for the model-driven development of advanced mechatronic systems as they are neither defined rigorously enough nor are appropriately tailored. The System Modeling Language (SysML) [35] which combines UML concepts with system engineering concepts, has the same limitations. It is mainly geared towards supporting early development phases but not the design phase which serves as a basis for code generation as in our approach.

The UML 1.5 profile Timed UML and RT-LOTOS Environment (TURTLE) enhances UML class and activity diagrams with concepts from RT-LOTOS and provides tool support for its validation by means of exhaustive simulation reusing existing tools for RT-LOTOS [4]. In contrast to this approach, MECHATRONIC UML supports verification even for large configurations by means of compositional reasoning and addresses hybrid behaviour.

Within the IST project AIT-WOODDES *hierarchical timed automata* (HTA) [14] are used to support the modeling and verification of complex real-time behaviour. HTA are a hierarchical extension of timed automata [26] and they provide for most of the powerful modeling concepts of statecharts as well as clocks like our HRTCs. However, no comparable concept for the definition of deadlines and no corresponding support for modelling component hierarchies exist in this approach. A mapping of HTA to multiple parallel running flat timed automata permits to verify the model by using the model checker UPPAAL [31], but does not support any compositional or modular reasoning. Code generation [3] is restricted to flat automata and does not take into account the delays that occur when transitions are fired. Our approach for code generation in contrast respects hierarchy, parallelism, and the real-time constraints.

In the OMEGA project [20], the UML has been extended by additional time constructs. A formal semantic definition of the extensions is currently under development. However, unlike our approach, there is no support for hybrid behaviour and compositional verification is only supported by semi-automatic verification via theorem proving.

Concerning the verification of adaptive behaviour, i.e. reconfiguration, only a few approaches exist so far. In contrast to [44] we suggest a compositional approach and integrate real-time and continuous behaviour by verifying our HRTSCs. Code generation from HRTSCs is also a unique feature of our approach. [21] only considers untimed models and assumes that a self-* capability of a system will fix certain types of problems in the long run, which do not have to be formally verified. In a safety-critical system this approach is not acceptable.

For a detailed comparison of tools supporting model-driven development of mechatronic systems we refer to [18].

9 Current and Future Work

So far, our approach allows constructing and verifying system structures with bilateral communication only, i.e. one-to-many or many-to-many communication structures (multilateral communication) are not supported. We are currently extending our approach by so-called multi-roles which enable to specify multilateral communication

protocols by parameterizing the described coordination patterns. In order to avoid model checking of possibly many parallel automata (in case the number of participating components becomes big), we define so-called regular communication structures. They define multilateral relationships between components as an iteration of bilateral communication, i.e. parameterized coordination patterns are again just the parallel composition of two automata.

The regular communication structures are again defined by a graph transformation system. The rules of such a system (as a further restriction to the rules described above) generate only component connections which have a corresponding (parameterized) coordination pattern defined. As an example, a forbidden graph structure in this case would incorporate multilateral component interactions without any corresponding coordination pattern [28].

We also work on the automatic synthesis of coordination patterns from scenario diagrams which specify particular use cases of the system under construction. This requires to define a consistent set of scenarios including timing constraints and an appropriate algorithm to generate coordination patterns. The specification of scenarios must be based on a well-defined set of component (types) together with possibly some behavioural restrictions. A first version of this approach has been published in [22].

As future work we see the extension of our verification framework to *parametric* systems, i.e. systems in which the number of components taking part in a reconfiguration is arbitrary. On the modelling side this is not difficult to achieve since there are variants of graph transformation rules which for instance allow for using universal quantification. On the verification side, this is less straightforward: the usual model checking approach requires a fixed, finite number of components as to be able to build the state space. The induction technique on the other hand requires a finite number of rules. Neither requirements are given for parametric systems. Here, we currently investigate the applicability of shape-analysis-like techniques [43, 41].

Another road we currently follow is to derive *plans* for safe reconfiguration as to be able to direct reconfiguration into some desired goal state.

Acknowledgements

We are indepted to the many people who contributed to MECHATRONIC UML and FUJABA. The results of many M.Sc. and PhD theses are very significant contributions to the project. In particular, Dr. Holger Giese, now with Hasso Plattner Institute at Potsdam University, was instrumental in developing the MECHATRONIC UML approach during his time he spent as assistant professor in the Software Engineering Group at Paderborn. Finally, we are grateful to Stefan Henkler and Dietrich Travkin for proofreading and helping out with editorial stuff.

References

1. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The Algorithmic Analysis of Hybrid Systems. In: 11th International Conference on Analysis and Optimization of Systems: Discrete Event Systems. Lecture Notes in Control and Information Sciences, vol. 199, pp. 329–351. Springer, Heidelberg (1994)

2. Alur, R., Dang, T., Esposito, J.M., Fierro, R.B., Hur, Y., Ivancic, F., Kumar, V., Lee, I., Mishra, P., Pappas, G.J., Sokolsky, O.: Hierarchical hybrid modeling of embedded systems. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 14–31. Springer, Heidelberg (2001)
3. Amnell, T., David, A., Fersman, E., Oliver Möller, M., Pettersson, P., Yi, W.: Tools for real-time UML: Formal verification and code synthesis. In: Workshop on Specification, Implementation and Validation of Objectoriented Embedded Systems (SIVOES 2001), Budapest, Hungary, pp. 1–4 (June 2001)
4. Apvrille, L., Courtiat, J.-P., Lohr, C., de Saqui-Sannes, P.: TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. IEEE Transactions on Software Engineering 30(7), 473–487 (2004)
5. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: Proc. of the 28^{th} International Conference on Software Engineering (ICSE), Shanghai, China, pp. 72–81. ACM Press, New York (2006)
6. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, pp. 125–126. IEEE Computer Society, Los Alamitos (2006)
7. Bender, K., Broy, M., Péter, I., Pretschner, A., Stauner, T.: Model based development of hybrid systems: Specification, simulation, test case generation. In: Modelling, Analysis, and Design of Hybrid Systems. Lecture Notes in Control and Information Sciences, vol. 279, pp. 37–51. Springer, Heidelberg (2002)
8. Briand, L.C., Wolf, A.L. (eds.): International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, Minneapolis, MN, USA, May 23-25 (2007)
9. Burmester, S., Giese, H., Gambuzza, A., Oberschelp, O.: Partitioning and Modular Code Synthesis for Reconfigurable Mechatronic Software Components. In: Bobeanu, C. (ed.) Proc. of European Simulation and Modelling Conference (ESMc 2004), Paris, France, pp. 66–73. EOROSIS Publications (October 2004)
10. Burmester, S., Giese, H., Henkler, S., Hirsch, M., Tichy, M., Gambuzza, A., Müch, E., Vöcking, H.: Tool Support for Developing Advanced Mechatronic Systems: Integrating the Fujaba Real-Time Tool Suite with CAMeL-View. In: Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA, pp. 801–804. IEEE Computer Society Press, Los Alamitos (May 2007)
11. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In: Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS 2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML 2004, pp. 1–20 (October 2004)
12. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent systems. International Journal on Software Tools for Technology Transfer 10(3), 207–222 (2008)
13. Burmester, S., Giese, H., Schäfer, W.: Code generation for hard real-time systems from real-time statecharts. Technical Report tr-ri-03-244, University of Paderborn, Paderborn, Germany (October 2003)
14. David, A., Möller, M.O., Yi, W.: Formal verification of UML statecharts with real-time extensions. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 208–241. Springer, Heidelberg (2002)
15. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars, pp. 247–312. World Scientific, Singapore (1997)

16. Flake, S., Mueller, W.: An OCL Extension for Real-Time Constraints. In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 150–171. Springer, Heidelberg (2002)
17. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA, pp. 179–188. ACM Press, New York (November 2004)
18. Giese, H., Henkler, S.: A survey of approaches for the visual model-driven development of next generation software-intensive systems. Journal of Visual Languages and Computing 17, 528–550 (2006)
19. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. In: Proc. of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International symposium on Foundations of Software Engineering (ESEC/FSE-11), pp. 38–47. ACM Press, New York (September 2003)
20. Graf, S., Hooman, J.: Correct Development of Embedded Systems. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 241–249. Springer, Heidelberg (2004)
21. Güdemann, M., Ortmeier, F., Reif, W.: Formal modeling and verification of systems with self-x properties. In: Yang, L.T., Jin, H., Ma, J., Ungerer, T. (eds.) ATC 2006. LNCS, vol. 4158, pp. 38–47. Springer, Heidelberg (2006)
22. Henkler, S., Greenyer, J., Hirsch, M., Schäfer, W., Alhawash, K., Eckardt, T., Heinzemann, C., Löffler, R., Seibel, A., Giese, H.: Synthesis of timed behavior from scenarios in the fujaba real-time tool suite. In: Proc. of the 31th International Conference on Software Engineering (ICSE), Vancouver, Canada (May 2009)
23. Henkler, S., Hirsch, M., Priesterjahn, C.: Hybrid model checking with the fujaba real-time tool suite. In: Aßmann, U., Johannes, J., Zündorf, A. (eds.) Proc. of the 6^{th} International Fujaba Days 2008, Dresden, Germany, pp. 40–43 (September 2008)
24. Henzinger, T.A.: The Theory of Hybrid Automata. In: Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS 1996), pp. 278–292 (1996)
25. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Hytech: The next generation. In: IEEE Real-Time Systems Symposium, pp. 56–65 (1995)
26. Henzinger, T.A., Manna, Z., Pnueli, A.: What Good Are Digital Clocks? In: ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
27. Hestermeyer, T., Oberschelp, O., Giese, H.: Structured Information Processing For Self-optimizing Mechatronic Systems. In: Araujo, H., Vieira, A., Braz, J., Encarnacao, B., Carvalho, M. (eds.) Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal, pp. 230–237. INSTICC Press (August 2004)
28. Hirsch, M., Henkler, S., Giese, H.: Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML. In: Proc. of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008), Leipzig, Germany, pp. 33–40. ACM Press, New York (May 2008)
29. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Computer society, Los Alamitos (2003)
30. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Briand and Wolf [8], pp. 259–268
31. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology 1(1) (1997)
32. Lynch, N.: Input/output automata: Basic, timed, hybrid, probabilistic, dynamic,.. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 191–192. Springer, Heidelberg (2003)

33. Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (September 2002), http://cgi.omg.org/docs/ptc/02-03-02.pdf
34. Object Management Group: Systems Modeling Language (SysML) Specification (January 2005)
35. Object Management Group: Systems Modeling Language (SysML) Specification (2006)
36. OMG: UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) (2008)
37. OMG: Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (September 2002), http://cgi.omg.org/docs/ptc/02-03-02.pdf
38. Osmic, S., Münch, E., Trächtler, A., Henkler, S., Schäfer, W., Giese, H., Hirsch, M.: Safe online-reconfiguration of self-optimzing mechatronic systems. In: Gausemeier, J., Rammig, F., Schäfer, W. (eds.) Selbstoptimierende mechatronische Systeme: Die Zukunft gestalten. 7. Internationales Heinz Nixdorf Symposium für industrielle Informationstechnik, pp. 411–426 (February 2008)
39. Priesterjahn, C., Tichy, M., Henkler, S., Hirsch, M., Schäfer, W.: Fujaba4eclipse Real-Time Tool Suite. In: Model-Based Engineering of Embedded Real-Time Systems (MBEERTS). LNCS, pp. 1–7. Springer, Heidelberg (2009) (to appear)
40. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
41. Rensink, A., Distefano, D.: Abstract Graph Transformation. Electronic Notes in Theoretical Computer Science 157(1), 39–59 (2006)
42. Schäfer, W., Wehrheim, H.: The Challenges of Building Advanced Mechatronic Systems. In: Briand and Wolf [8], pp. 72–84
43. Wilhelm, R., Reps, T.W., Sagiv, S.: Shape analysis and applications. In: The Compiler Design Handbook, pp. 175–218 (2002)
44. Zhang, J., Cheng, B.H.C.: Model-based Development of Dynamically Adaptive Software. In: ICSE 2006: Proceeding of the 28th International Conference on Software Engineering, New York, NY, USA, pp. 371–380 (2006)

Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent

Holger Giese, Stephan Hildebrandt, and Stefan Neumann

Hasso Plattner Institute for Software Systems Engineering
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
{Holger.Giese,Stephan.Hildebrandt,Stefan.Neumann}@hpi.uni-potsdam.de

Abstract. During the overall development of complex engineering systems different modeling notations are employed. For example, in the domain of automotive systems system engineering models are employed quite early to capture the requirements and basic structuring of the entire system, while software engineering models are used later on to describe the concrete software architecture. Each model helps in addressing the specific design issue with appropriate notations and at a suitable level of abstraction. However, when we step forward from system design to the software design, the engineers have to ensure that all decisions captured in the system design model are correctly transferred to the software engineering model. Even worse, when changes occur later on in either model, today the consistency has to be reestablished in a cumbersome manual step. In this paper, we present how model synchronization and consistency rules can be applied to automate this task and ensure that the different models are kept consistent. We also introduce a general approach for model synchronization. Besides synchronization, the approach consists of tool adapters as well as consistency rules covering the overlap between the synchronized parts of a model and the rest. We present the model synchronization algorithm based on triple graph grammars in detail and further exemplify the general approach by means of a model synchronization solution between system engineering models in SysML and software engineering models in AUTOSAR which has been developed for an industrial partner.

1 Introduction

The development of complex engineering systems involves different modeling notations from different disciplines. Taking the domain of automotive systems as an example, SysML (System Modeling Language) [1] models are employed quite early to capture the requirements and basic structuring of the whole system by system engineers, while AUTOSAR (Automotive Open System ARchitecture)[1] models are used later in the software development process to describe the

[1] http://www.autosar.org

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 555–579, 2010.

concrete software architecture and its deployment. Using these different models helps in addressing each specific design issue with an appropriate notation and at a suitable level of abstraction.

For example, when going from the system design with SysML to the software design stage with AUTOSAR, today the engineers have to ensure manually that all relevant decisions captured in the SysML model are correctly transferred to the AUTOSAR model. When changes occur later on either in the AUTOSAR or SysML model, the situation is even worse: The consistency has to be reestablished in a cumbersome manual step that inspects both models and transfers all detected changes. Otherwise, the integration of the different system parts as captured by the SysML model and refined in the AUTOSAR model may fail.

Model-Driven Engineering (MDE) with its support for model transformation and model consistency checking is a promising direction to approach the sketched model consistency problems, which result as the models describe the system under development from different viewpoints and on different levels of abstraction capturing only partially overlapping information (cf. [35]).

Triple graph grammars (TGGs) are a formalism to declaratively describe correspondence relationships between two types of models. They were introduced in [29] and are one option to specify the required model transformations using a declarative transformation specification. In several contexts, different variants of TGGs have already been employed for model synchronization such as the integration of SysML models with Modelica simulation models [22], keeping models from the domain of chemical engineering consistent [4] and transformations from SDL models to UML models and vice versa [7].

In this paper, we report about our approach to tackle the outlined model synchronization problem. Built on top of techniques from model-driven engineering such as meta-models, consistency rules, and bidirectional model transformations resp. model synchronizations specified by TGGs, a general architecture has been developed, which allows to automate the task of keeping models consistent. We only require that the TGG rule set is deterministic and that only one of the models is changed at a time. Like described in [13], in many cases managing and tolerating inconsistencies (e.g., by allowing to manipulate different models concurrently) instead of directly removing them is desirable. In the case of the automotive domain, consistency plays a crucial role, e.g., caused by the reason that inconsistencies between previously defined requirements and the later implementation can lead to catastrophic failures. Thus, a more rigorous handling of inconsistencies like requested in [13] is adequate for our application example.

In a project with the automotive industry, we could demonstrate that our approach can be employed for model synchronization between the SysML tool TOPCASED and the AUTOSAR tool SystemDesk. Firstly, the model transformation derived from TGGs permits to automatically generate the initial AUTOSAR model from the SysML model. Secondly, consistency between both models in case of changes in one of them can be maintained by a TGG-based model

synchronization system [19]. Thus, we can synchronize both models such that changes within one are automatically transferred to the other.[2]

By making manual transformation and synchronization steps obsolete, the automatic synchronization of models reduces costs and time. This applies to the initial transitions, for example, from the SysML model to the AUTOSAR model, as well as the re-establishment of consistency in case of changes in one model. In addition, such automated synchronization steps are less error prone than manual steps as employed today. They further enable employing iterative and more flexible development processes as the costs for iterations or changes are dramatically reduced as long as parallel changes do not occur.

The structure of the paper is as follows: The current state of the art and its limitations compared to our outlined new approach are discussed in Section 2. The considered application example for our approach for synchronizing SysML and AUTOSAR models is introduced in Section 3. The new approach, its architecture, and its components are first sketched in Section 4. Then the constituent parts are presented in detail. The technique for model synchronization is explained in Section 5. The tool adapter and the techniques for consistency checks follow in Section 6 and Section 7. At the end, we discuss the suitability of our approach by looking into several typical usage scenarios, such as the initial transfer of information or change propagation, and close the paper with a final conclusion and an outlook on planned future work.

2 State of the Art

In this section, we discuss related approaches for model transformation and synchronization, and related work in the context of Model-Driven Engineering, that make use of model transformations.

2.1 Model Synchronization

An overview of model transformation and synchronization systems can be found in [9]. The paper categorizes existing approaches and briefly explains them.

As outlined in the introduction, MDE requires a bidirectional solution, which preserves model contents when synchronizing as much as possible. However, many available model transformation approaches only support classical one-way batch-oriented transformations [16]. The QVT implementations [6] and [15], and some graph transformation-based approaches such as VIATRA [33,5], the GREAT model transformation system [34], AGG [12], the core PROGRES tool [28] or the core FUJABA tool [32] are only unidirectional but partly incremental. The available relational QVT implementations [21,31] as well as BOTL [25] are bidirectional but only support a batch-oriented processing and, thus, fail to be scalable.

[2] It has to be noted that the required restriction concerning no parallel changes in the models does not result in any additional limitations in the considered application domain, as the processes currently try to exclude changes at all to avoid the cumbersome manual step to reestablish consistency.

Other existing TGG-based approaches also do not provide a comparable automatic and computational incremental solution (for a detailed discussion see [19]): The TGG transformation algorithm based on the PROGRES environment [4] is also incremental, but operates interactively, and is therefore inappropriate for the transformation of large models. In the incremental TGG transformation approach supported by ATOM^3 [20], updates are triggered by user actions like creating, editing or deleting elements and the specification of updates for all possible user actions is required. Thus, the consistency of the approach is difficult to guarantee and initial complete model transformations are not supported. Another TGG realization based on [32] is MOFLON [10]. It focuses on model integration and transformation for the MOF 2.0 standard [26] rather than incremental model synchronization.

Incremental model synchronization can also be seen as an inconsistency resolution problem. [11] describes an incremental solution for the related problem of inconsistency checking. The presented system allows a check to be made to quickly determine whether a modification has caused inconsistencies, and proposes solutions to the user. For a more detailed discussion of such solutions for model synchronization we refer to [19].

2.2 Model Integration

Model-Driven Engineering is a development paradigm, where models are the primary development artifacts. In [23] this idea is described. Models are used to describe the system under development from different viewpoints and on different levels of abstraction. During the development process, models are refined and ultimately source code is generated from these models. The use of different kinds of models leads to the problem of keeping those models consistent to each other. At this point, model transformation systems play a central role. In practice an additional challenge is that different kinds of models are normally supported by different tools and these tools use diverse technologies for representing these models. So at first models need to be accessed in an appropriate way to be able to apply model transformation techniques.

In the MATE project [30] an adapter has been realized to access MATLAB/Simulink models in such a way that model transformation rules can be applied, e.g., for checking guidelines while model consistency like in case of model synchronization is not the main focus.

In the ModelBus project [2] a framework has been developed, which is able to integrate different model-based development tools into a service-oriented middleware. The purpose of the ModelBus project is to provide a framework allowing several tools to be connected within a single environment. Model transformation and synchronization techniques can be potentially applied using this framework. When access to the different models is provided in an adequate form, the integration of different models using model transformation and synchronization techniques can be realized.

An approach for the integration of SysML models with Modelica simulation models has been described in [22]. The approach is also based on triple graph

grammars but uses VIATRA [8] to implement the transformation. In contrast to the system presented in this paper, synchronization of models is not supported.

3 Application Example

We have evaluated the approach presented in this work within a project organized with our industrial partner, dSPACE GmbH. dSPACE provides, besides other products, several tools for the development of embedded systems, especially for the automotive domain. Within this project we used two different modeling languages commonly used for the development of automotive embedded systems, namely the *System Modeling Language* (SysML) and the *AUTomotive Open System ARchitecture* (AUTOSAR). We used the tool SystemDesk, a professional tool for the development of complex automotive embedded systems according to the AUTOSAR standard, and TOPCASED, an open-source toolkit for supporting the modeling of SysML models. Both modeling languages (AUTOSAR and SysML) are subsequently described in more detail.

3.1 SysML

A widely used language for system engineering is SysML (System Modeling Language), which is currently available in version 1.1 (see [1]). SysML supports the design and analysis of complex systems including hardware (HW), software (SW), processes and more. SysML reuses a subset of the UML and adds some additional parts (e.g., the Requirement and Parametric Diagram) to facilitate the engineering process by providing several improvements compared to UML concerning system engineering. UML itself tends to be more software-centric while the topic of SysML is clearly set to the analysis and design of complex systems (not only SW).

In our application example the existing SysML profile provided by the OMG has been used, which utilizes the generic extension mechanisms of UML to customize UML elements using the concept of stereotypes. Such stereotypes can be applied to UML elements[3] and extend as well as define constraints on these elements. For expressing constraints also the *Object Constraint Language* (OCL) can be used, which not only allows the description of structural properties, but also the specification of additional constraints on the values and/or types of attributes and so on. For more information about OCL see [27]. Instances of the stereotyped UML elements must fulfill the properties defined by the applied stereotypes. The SysML profile (like any other UML profile) contains a set of stereotypes, which are applied to a UML model. In the following we explain relevant SysML stereotypes for a small application example.

In SysML, system blocks are used to specify the structure of the system[4]. For this purpose the UML meta model element *Class* is extended by the stereotype *<<block>>*. A block describes a logical or physical part of the system

[3] Elements of the UML meta model.

[4] A block describes a part of the structure of an interconnected system.

(e.g., SW or HW). Multiple of these blocks can be used for representing the structure of a system. An example for the additional capabilities of SysML is the possibility to model the flow of objects between different system elements (which are specified in form of SysML blocks) by using *<<flow ports>>*. *<<flow port>>* is a stereotype for the UML element *Port* and allows the modeling of an object flow between SysML blocks. For the specification of objects and data, which flow over a flow port, the stereotype *<<flow specification>>* is applied to the UML element *Interface* in SysML. Ports can be connected via connectors provided by the UML meta model. The elements required to connect different ports (the UML *Connector* and *ConnectorEnd*) as well as a part of the SysML meta model describing blocks, flow ports and flow specifications are shown in Figure 1 in a simplified way.

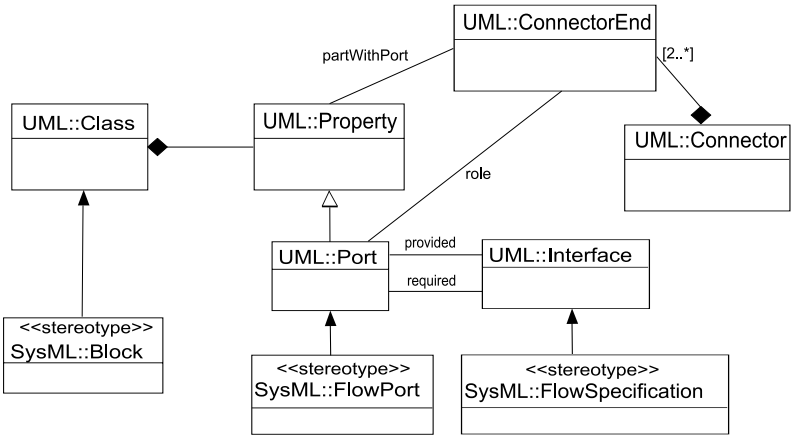


Fig. 1. Extract of the SysML metamodel

When analyzing and designing automotive systems, the HW/SW-structure can be described using SysML blocks, ports (e.g., flow ports) and appropriate interfaces (e.g., flow specifications). In this paper, we use a simplified version of the structural constituents taken from an application example of an engine-fuel control system consisting of actuators and sensors for the throttle position and the control software. The control software evaluates the sensor values, computes appropriate throttle position values and sends them to the actuator of the throttle.

The system structure including HW and SW parts has been modeled using the tool TOPCASED[5] and the resulting SysML model of the engine fuel control system is shown in Figure 2. The example consists of six different types of blocks, three of them represent hardware parts like the engine, a HW actuator and a HW sensor for setting and measuring the throttle position of the engine.

[5] http://www.topcased.org/

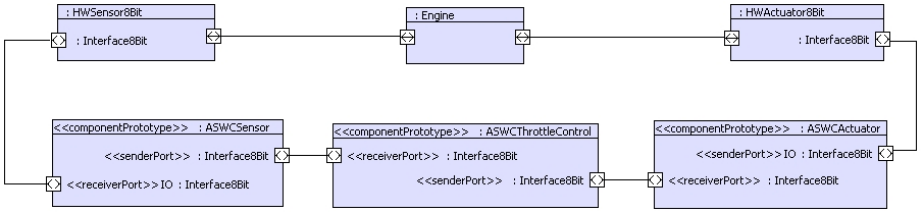


Fig. 2. Application example of an SysML model created in Topcased

The HW sensor (*HWSensor8Bit*) is connected to a SW block (*ASWCSensor*), which reads data from the HW (e.g., by using driver functionality) and sends these measured values to a SW block, which realizes the control functionality (*ASWCThrottleControl*) and computes an output signal. This output signal is send to a SW block (*HWActuator*), which realizes the access to the HW actuator, which is represented through the block *HWActuator8Bit*. The *HWActuator* interacts with the representation of the physical engine.

When such a system is designed, several restrictions have to be considered concerning the used HW sensor blocks in combination with the software blocks. A typical restriction is that a connector can only connect ports, which implement the same interface. In the shown example, e.g., the flow ports of the blocks *ASWCSensor* and *HWSensor8Bit*, over which these two blocks are connected, have to implement the same interface. Such a constraint can be expressed in the form of the following OCL constraint for the type connector:

```
context Connector inv:
self.end->forAll(e:self.end->get(0).role.type = e.role.type)
```

Only three of the blocks (*ASWCThrottleControl*, *ASWCSensor* and *ASWCActuator*) described above are relevant for the SW architecture. In our implementation, stereotypes are defined for identifying, e.g., the definition of SW blocks (<<*atomicSoftwareComponent*>>) as well as for the usage of the defined SW blocks (<<*componentPrototype*>>) like shown in Figure 2. In the following section, we show how these constituents can be represented in AUTOSAR.

3.2 AUTOSAR

AUTOSAR (Automotive Open System ARchitecture) is a framework for the development of complex electronic automotive systems. The purpose of AUTOSAR is to improve the development process for ECUs (Electronic Control Units) and whole systems by defining standards for the system and software architecture. The AUTOSAR standard defines a meta model, which describes a DSL for the development of automotive embedded systems. The part of the meta model relevant for the present work is described in [3] in the form of a UML profile. We use a stand-alone meta model for AUTOSAR, which is realized accordingly.

As defined by the AUTOSAR meta model (an excerpt is shown in Figure 3), the software architecture is built from components (e.g., *AtomicSoftwareComponents* (ASWC)). These ASWC are derived from the type *ComponentType* and can communicate using two different categories of ports: required and provided ports (represented through *RPortPrototype* and *PPortPrototype*). Both types are derived from the same abstract class *PortPrototype*. An *RPortPrototype* only uses data or events, which are provided by other ports of type *PPortPrototype*. A port of type *RPortPrototype* or *PPortPrototype* can implement an interface of type *PortInterface*. This *PortInterface* is refined by *ClientServerInterface* and *SenderReceiverInterface*.

The SW blocks (*ASWCSensor*, *ASWCActuator* and *ASWCThrottleControl*) defined within the SysML model described above can also be specified within an AUTOSAR model. The blocks shown in Figure 2 can also be described using ASWCs, ports and interfaces, which are defined within the extract of the AUTOSAR meta model shown in Figure 3. Figure 4 shows the same SWCs modeled with the tool SystemDesk[6].

In case of the SysML example, the SW blocks, ports and connectors can be described directly within such an AUTOSAR model in the form of ASWCs. In case of the blocks describing HW, such a mapping is not desired by our industrial partner. HW components in AUTOSAR are represented on completely different levels of abstraction than in SysML. Therefore, the blocks, ports and connectors concerning HW in the SysML model have not been reflected in the AUTOSAR model in our application example. Also the connectors, which exist in the SysML model between the ports of a SW block and a HW block have not been transformed to AUTOSAR.

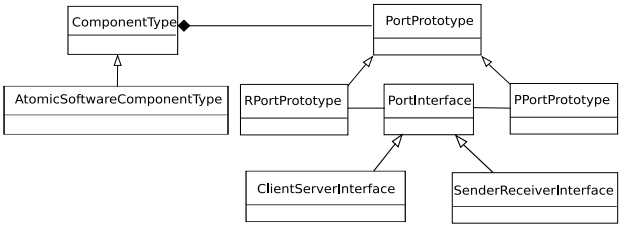


Fig. 3. Extract of the AUTOSAR meta model

3.3 Common Constituents

The elements in Figure 2 are tagged with stereotypes, e.g., <<*senderPort*>> and <<*receiverPort*>>. This is necessary because there are different types of ports in AUTOSAR but only one type in SysML. Another example are software components. AUTOSAR supports atomic and composite software components.

[6] http://www.dspace.com/ww/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm

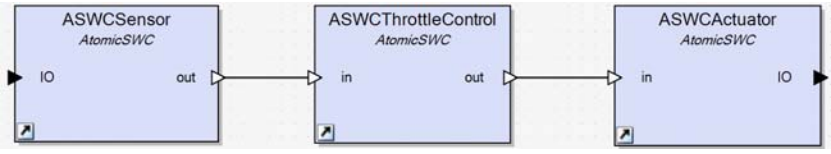


Fig. 4. AUTOSAR model derived from the SysML model

Both are represented as blocks in SysML. To distinguish both types, the stereotypes *<<atomicSoftwareComponentType>>* and *<<compositionType>>* have to be used. These stereotypes have been defined in a small profile for SysML, which we created for this purpose. In the remainder of this work we describe how to support consistency of such semantically identical elements in different models using transformation and synchronization techniques.

4 Approach

4.1 General Architecture

The generic architecture to integrate model transformation and synchronization with existing modeling tools is shown in Figure 5. The transformation system only supports EMF-compatible (Eclipse Modeling Framework [7]) models. Therefore, the source and target models need to be provided in that format. This is done by tool adapters, which translate the models from and to EMF if necessary. If the modeling tool itself is based on EMF, such an adapter can be realized easily. If the modeling tool is not based on EMF, the tool adapter has to provide an EMF representation on the fly that the transformation system can modify, and has to synchronize it with the actual model in the modeling tool. More information on how such an adapter could be realized can be found in Section 6.

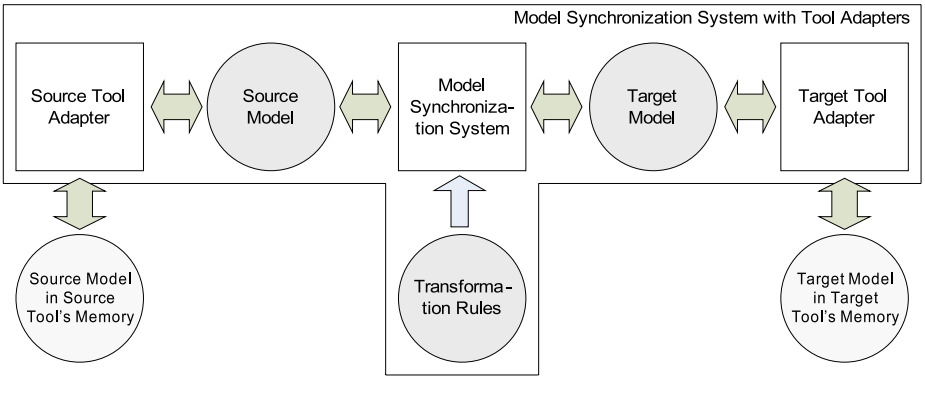


Fig. 5. General Architecture

[7] http://www.eclipse.org/modeling/emf/

4.2 Architecture Example

In the project realized with our industrial partners, we have established an architecture like described above, which integrates the tools TOPCASED and SystemDesk. For each tool an adapter has been realized, that provides an EMF representation of the model to the transformation system. Figure 6 shows the concrete architecture developed in that project.

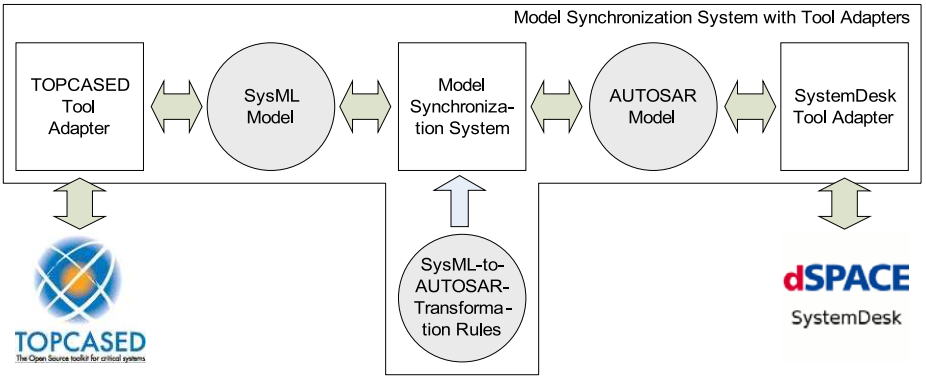


Fig. 6. Integration architecture for TOPCASED-SystemDesk integration

The transformation system, the tool adapters, and TOPCASED are based on the Eclipse platform. TOPCASED already uses EMF as its underlying modeling infrastructure. Therefore, access to these models from the transformation system can be realized without great effort. SystemDesk is a stand-alone Windows application that provides a COM interface for access to its models. Accessing SystemDesk's models is more difficult, because the technology gap between Eclipse/EMF and SystemDesk/COM must be bridged. For this purpose, we have developed a dedicated adapter. While the transformation system creates and modifies an AUTOSAR model in EMF representation, the SystemDesk adapter takes care of reading and writing the model to and from SystemDesk. For more details concerning the realization of the SystemDesk adapter see Section 6.

5 Model Synchronization System Based on Triple Graph Grammars

The model transformation system is based on triple graph grammars [29]. It is able to perform model transformations in both directions, i.e. create a target model from a source model and vice versa. Furthermore, it can synchronize both models after modifications have occurred. In the following sections, triple graph grammars are briefly introduced and the transformation and synchronization algorithm is explained.

5.1 Triple Graph Grammars

Triple graph grammars combine three conventional graph grammars to describe the correspondence relationships between elements of two types of models. Two graph grammars describe the two models and a third grammar describes a correspondence model. Figure 7 shows a TGG rule for the transformation of a SysML block to an atomic software component in AUTOSAR. This illustration also combines the left-hand side (LHS) and right-hand side (RHS) of the rule. The black elements belong to the LHS and the RHS of the rule, i.e. they form the application context. The elements marked with ++ (and printed green) belong only to the RHS and are created when the rule is applied. Rules that delete elements are not used in the context of model transformation with TGGs (cf. [29]). The correspondence model is used to explicitly store correspondence relationships between corresponding source and target elements. It allows the target model elements corresponding to a given source model element to be found quickly. The correspondence nodes are connected to each other and form a directed acyclic graph. Although, there is no link visible in Figure 7 between *CorrPackage* and *CorrASWC*, that link is created implicitly and is not shown in the TGG rule to ease modeling of TGG rules.

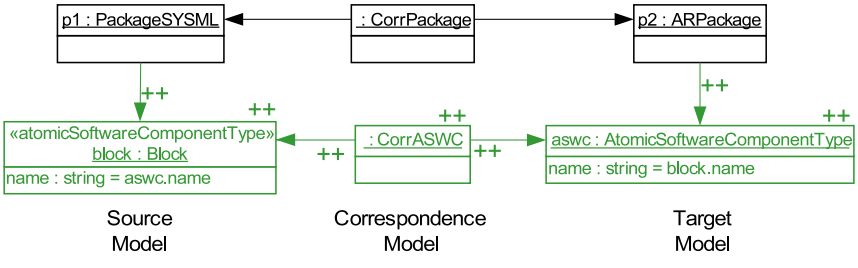


Fig. 7. TGG rule for the transformation of a block to an atomic software component

Fig. 8. Axiom of the triple graph grammar for the transformation from SysML to AUTOSAR

Like every other graph grammar, a triple graph grammar has a start graph which serves as the starting point of the transformation. In the context of TGGs it is called *axiom*. Figure 8 shows the axiom for the transformation from SysML to AUTOSAR. The whole grammar for this transformation contains many more rules, which are not shown here due to space limitations.

5.2 Model Transformation

TGG rules are declarative by nature. To execute them they can be either interpreted by a dedicated TGG interpreter like presented in [24], or other executable artifacts can be derived. In our case, Story Diagrams [14][8] are generated, which are executed by a Story Diagram interpreter [18].

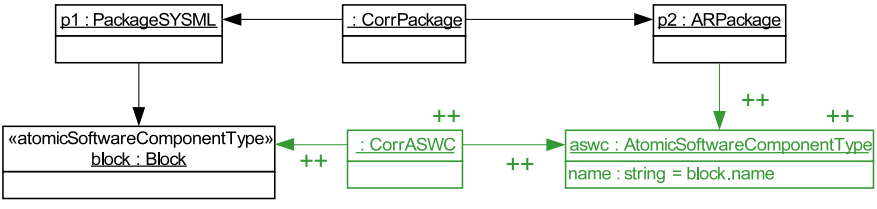


Fig. 9. Operational rule for the forward transformation of a block to an atomic software component

The transformation system consists of two major parts, the transformation engine and the operational transformation rules. The engine is independent from specific source or target models and invokes the rules. The transformation system supports both transformation directions, i.e. creating the right model from the left model and vice versa. Furthermore, synchronization of models is also supported. This is explained in Section 5.3. Therefore, separate operational rules for each direction are required. Conceptually, the operational rules are derived by adding all elements on the source model side to the rule's application context. Figure 9 shows the conceptual forward transformation rule derived from the rule in Figure 7. In practice, the operational rules have to do much more. Therefore, four separate operations are generated for each rule and for each direction, that perform

1. transformation of elements (*transformation()*)
2. synchronization of elements (*synchronization()*)
3. synchronization of attributes (*synchronizeAttributes()*, called by 2)
4. reconstruction of broken structures (*repairStructure()*, called by 2)

The overall operation principle of the transformation engine and the *transformation()* operation is explained in the following, the others are described in the next section.

The operation principle of the transformation engine is depicted in Figure 10. To execute a model transformation, the engine is started with the root elements of the source and target models as parameters, as well as the desired transformation direction (i.e. forward or backward). First, the axiom is executed to transform the root node (1). The correspondence node that was created by

[8] Story Diagrams combine UML activity diagrams with graph transformation rules to describe behavior.

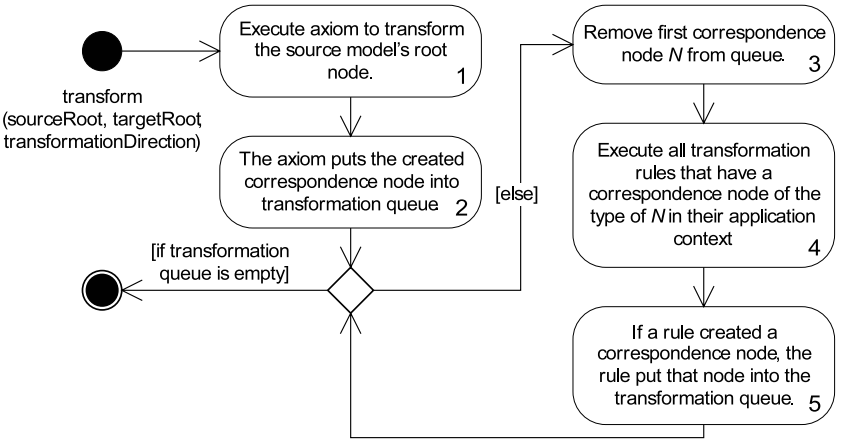


Fig. 10. Operation principle of the model transformation engine

the axiom is put into the transformation queue of the engine (2). This step is done by the axiom itself. The transformation queue contains the correspondence nodes that need to be processed. After that, the first correspondence node in the queue is removed (3) and all transformation rules are executed that expect such a correspondence node in their application contexts (4). If a rule has successfully transformed an element, the associated correspondence node is put into the transformation queue (5). Steps 3 to 5 are repeated until the transformation queue runs empty. Then the transformation is complete.

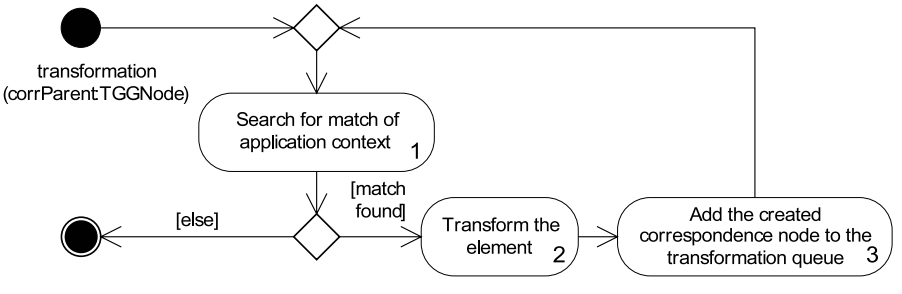


Fig. 11. Operation principle of the *transformation()* operation

Figure 11 shows the operation principle of the *transformation* operation[9]. The operation's parameter is the parent correspondence node (in case of the rule in Figure 9, the *CorrPackage* node). This node is the starting point of the search for other elements of the application context of the rule (*PackageSYSML*, *ARPackage* and *Block* in Figure 9). If these elements can be found and if they were

[9] Axioms are only special kinds of rules. Therefore, the operation principle of axioms is virtually the same.

not transformed before (1), the correspondence node and target elements are created according to the TGG rule (2). After that the newly created correspondence node is added to the transformation queue of the transformation engine (3). This process is repeated as long as new matches for the rule's application context can be found. After that, the control flow returns to the transformation engine as described above.

5.3 Model Synchronization

The model transformation system can also perform a synchronization between the models after an initial transformation. For efficiency, the system only visits those nodes that were actually modified. To detect modifications, an event listener is registered at each element of the source and target models. If an element is modified, its associated correspondence node is put into the transformation queue. There is an additional flag associated with each node in the queue[10], that marks whether the consistency of a correspondence node should be checked (flag is true), or new elements should be transformed (flag is false) when this correspondence node is processed by the engine. The notification listener always sets this flag to true. The transformation rules (see Section 5.2, step 3) always set it to false.

Therefore, the actual operation principle of the transformation engine is slightly more complicated than described in Figure 10. In step 4 of Figure 10, first the flag is checked. If the flag is false, the *transformation* operations are executed. If the flag is true, only the *synchronization* operation (see below) is executed that belongs to the rule that created the current correspondence node. The synchronization rule is responsible for re-establishing consistency between the associated source and target model elements. Furthermore, in case of synchronization, the axiom is not executed beforehand because the root model elements already exist.

The synchronization operations are used in an attempt to preserve existing target elements as far as possible. Many modifications, like moving elements, can be synchronized by adjusting some links in the target model instead of deleting and retransforming elements. This reduces the number of necessary modifications in the target model and allows even large models to be synchronized very quickly in many cases.

Figure 12 depicts the operation principle of a synchronization rule. First, the rule checks if the structure of the source, target and correspondence elements complies with the rule. If the structure is valid, the *synchronizeAttributes()* operation is called (1). This operation compares the attribute values, synchronizes them if necessary, and returns whether attribute synchronizations were performed. If attribute values were actually modified, the subsequent correspondence nodes are put into the queue with their flags set to true to check their consistency as well (2). If no attribute synchronizations were necessary, the subsequent correspondence nodes do not need to be checked. In any case, the current

[10] For simplicity, this has been omitted in Section 5.2.

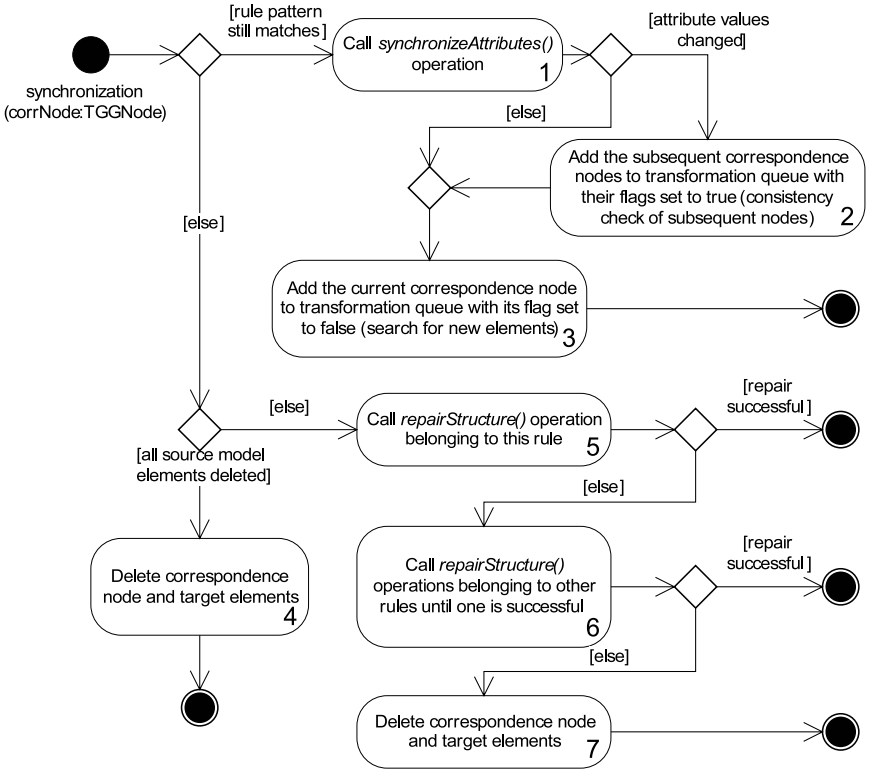


Fig. 12. Operation principle of the *synchronization()* operation

correspondence node is also added to the transformation queue with its flag set to false to search for new elements to transform (3).

In case the rule pattern does not match, a check is made to determine whether all source elements of the correspondence node were deleted from the model. Then the correspondence node and the target model elements are obsolete and can be deleted, as well (4). Note that this implies that all subsequent correspondence nodes and their target elements have to be deleted, too. If at least one of the source elements is still part of the model, a repair is attempted. First, the *repairStructure()* operation (see below) belonging to the same rule is executed (5). In case the repair is successful, the operation is finished. If it fails, the repair operations of the other rules are tried (6). As soon as one of them succeeds, the operation terminates. Note that the *repairStructure()* operation adds correspondence nodes to the transformation queue if necessary. Should all repair attempts fail, the correspondence and target elements are deleted (7). The modified source element cannot be synchronized with the available transformation rules.

The *repairStructure()* operation is the key to minimizing the number of write operations on the target model to re-establish consistency. Its general operation principle is shown in Figure 13. Prerequisites for a successful repair are that

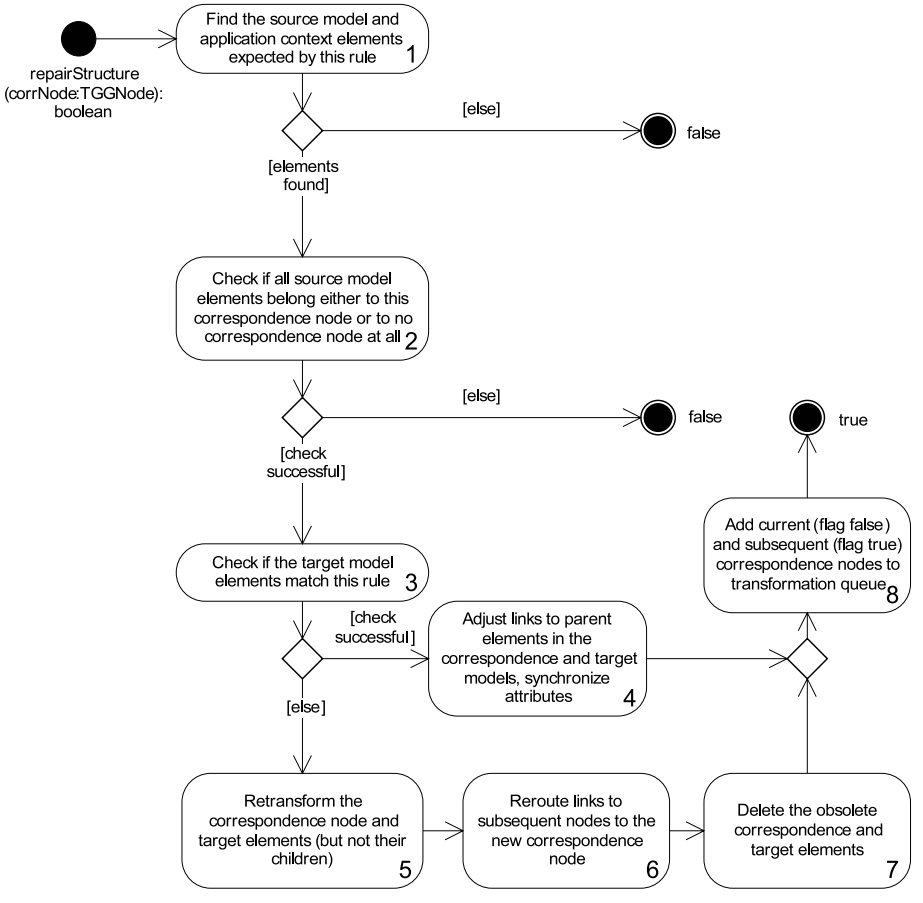


Fig. 13. Operation principle of the *repairStructure()* operation

the source model and application context elements exist (1) and that the source model elements are not connected to any other correspondence nodes (2).[11] If these conditions are met, the target element pattern is checked (3). If it complies with this rule, consistency can be re-established easily by re-adjusting the links to the elements of the application context of the rule (4). For the rule in Figure 7 this means to delete the existing link between *aswc* and the *ARPackage* it is currently linked to, and create a new link between *aswc* and the *ARPackage* that was matched in step 1. Moved elements can be synchronized mostly with this simple repair action.

More complex modifications can lead to the applicability of a different rule. In this case, the target elements do not meet the expected pattern because they were created by another rule. Then the correspondence node and target elements

[11] If connected, this means that these elements were already transformed by a different rule.

have to be created according to that rule (5). The links to subsequent correspondence nodes are rerouted to the newly created correspondence node (6), and the obsolete correspondence node and target elements are deleted (7). Finally, the current (or newly created) correspondence node is added to the transformation queue with its flag set to false to search for new elements to transform. Also the subsequent correspondence nodes are added to the transformation queue with their flags set to true to check their consistency. Depending on whether the structure could be repaired or not, true or false is returned by this operation.

This synchronization algorithm has some major advantages. First, the repair of broken structures minimizes the number of required write operations on the target model to synchronize modifications. While the previous version of our algorithm ([19]) would discard the target elements and retransform them, additional details that are not reflected in the source model would be discarded, as well. By preserving target elements, those details are preserved, too. This is important if the connected models have different levels of detail. Of course, also the performance is much higher if only links are changed and elements are not recreated. Second, the synchronization starts directly at those correspondence nodes where a modification took place. Those parts of the models that were not changed are not checked by the system. In addition, the synchronization stops checking correspondence nodes if a modification does not have any effects on them. For example, moving an element in the model usually does not influence its child elements. The algorithm first synchronizes the movement of the parent element. Then, its direct children are checked. If they have not been affected by the modification, the remaining indirect children are not checked. All in all, these optimizations make synchronization effort mostly independent from the overall model sizes. The size of the modifications (number and severity of modifications) has the largest impact on synchronization performance.

6 Tool Adapter

The tool adapter already mentioned in Section 4 is responsible for providing access to a modeling tool's in-memory model to the transformation system. Of course, the modeling tool has to provide a means to access its model, e.g. a COM interface. This allows models to be synchronized without indirection via files.

The transformation system works only on EMF-based models. Therefore, the tool adapter has to provide an EMF-based model. If the modeling tool is also based on EMF, like TOPCASED, such an adapter is quite simple. It only has to get the model's root element from the model editor and provide it to the transformation system (left side of Figure 6). The transformation and synchronization take place directly on the model.

If the modeling tool is not based on EMF or not even on the Eclipse platform, a tool adapter becomes much more complicated. It has to provide an EMF-based version of the model to the transformation system in addition to the model of the modeling tool. Before and after the transformation system reads or writes the EMF-based model, the adapter has to synchronize it with the model in the

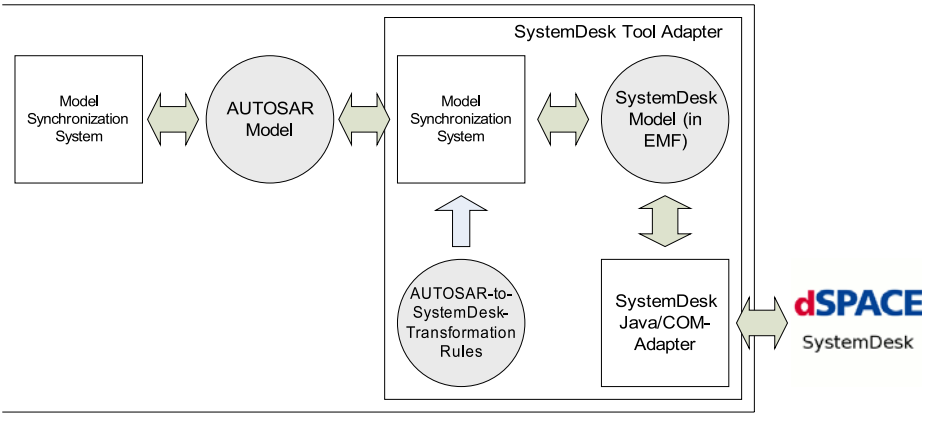


Fig. 14. Structure of the SystemDesk adapter

external modeling tool. This problem is explained in the following using *dSPACE SystemDesk* as an example.

dSPACE SystemDesk is a proprietary modeling tool for AUTOSAR modeling. It offers a *COM/.NET* interface to access the model that is currently loaded in SystemDesk. Therefore, a special *Java/COM* adapter is required to connect the Java-based transformation system to SystemDesk. The architecture of the SystemDesk adapter is shown in Figure 14. Another problem is SystemDesk's meta model, which is different from the AUTOSAR meta model. For example, SystemDesk provides a *Project Library* that contains the types of software components, ports, etc. Instances of these types can be used in the system model. However, it is also possible to create a type directly in the system model. Furthermore, a SystemDesk model contains several predefined root packages for the software, hardware and system configurations. These correspond to ordinary compositions in AUTOSAR which are not contained in any other composition. This problem is aggravated because SystemDesk's meta model was not available to us, and had to be reconstructed from the COM interface specification.

The tool adapter has to handle these SystemDesk specific issues and produce a standard-conformant EMF-based AUTOSAR model. First, we have tried to do the translation directly in the adapter code [17]. However, this has led to a very complex and hardly maintainable adapter code. While this transformation is essentially a model transformation from a SystemDesk to an AUTOSAR model, we have now used the model transformation system a second time to perform this transformation. The COM interface is used to synchronize SystemDesk's model with a corresponding EMF-based model. The model transformation system synchronizes this model with the AUTOSAR model that the adapter provides to the transformation system. This solution makes maintenance of the adapter much easier. Most of the adapter's logic is encoded in model transformation rules that can easily be adapted and extended. Maintainability is an important issue due

to the enormous complexity of the AUTOSAR meta model and its constant advancement.

We have encountered another problem, which regards object identity. It is not possible to directly reference an object in SystemDesk's memory. Instead a corresponding Java object has to be created. In an earlier version, SystemDesk did not support UUIDs. Therefore, it was hard to match Java objects to SystemDesk objects. This problem has been circumvented by always creating a second EMF-based copy of the current SystemDesk model, comparing it to the first copy, and merging the differences into the first copy using EMF Compare[12]. However, EMF Compare does not work very reliably without UUIDs. Now, SystemDesk supports UUIDs and matching corresponding objects in Java and SystemDesk is easy.

7 Model Consistency

Model consistency plays an important role, not only regarding consistency between different models or model elements of different models, but also between the elements of one model. In the following, we describe why model consistency is a crucial aspect, especially when model synchronization techniques are applied.

When model synchronization techniques are used, normally several model elements of two different synchronized models describe the same thing. These model elements can be synchronized using model synchronization techniques like TGGs. Model consistency concerning these semantically identical elements of different modeling languages is supported by the synchronization itself, as changes of model elements included in one model are carried over to the other model.

In most cases where synchronization techniques are used, the property holds that not all elements of a model are also reflected in a corresponding synchronized model. Normally, this is the case because different modeling languages, and consequently also different types of models, are synchronized. Such modeling languages have a specific purpose and, thus, different properties with different semantics are expressed by different languages. Therefore, dependencies or other properties can exist within the same model between synchronized and non-synchronized elements, which are not reflected by the synchronization mechanism itself.

In Figure 15, the two big circles on the left and right side represent two different models. In each model, a subset (represented through a cloud) is synchronized via a model synchronization system with the semantically identical elements of the corresponding model. Thus, consistency between elements from the cloud of the left side and on the right side is maintained by the synchronization system itself. Like denoted by the arrow with the exclamation mark on top, dependencies can also exist within the same model between synchronized and non-synchronized elements. Such properties can be invalidated when a model element is updated by a synchronization activity.

[12] http://www.eclipse.org/modeling/emft/

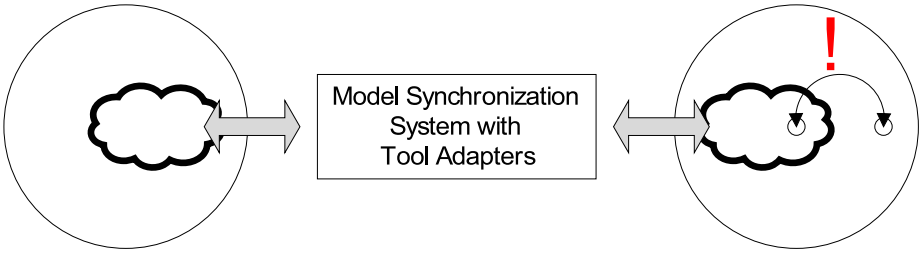


Fig. 15. Synchronization of parts of the models can lead to inconsistencies with other parts

An example of such a situation, where additional properties need to be checked is described subsequently. The SysML model shown in Figure 2 consists of six SysML blocks, but only the three lower blocks represent software. In Figure 4, these three SysML blocks are reflected in the form of semantically identical elements of an AUTOSAR model. The other three SysML blocks, which represent hardware, are not present in the AUTOSAR model. In case these two models are synchronized, only the constituents representing software are synchronized between the SysML and AUTOSAR model. Changing the bit width of the *IO* port of the *ASWCSensor* of Figure 4 is a valid operation in the AUTOSAR model, but applying these changes via the synchronization to the corresponding SysML model leads to a violation of a property of the SysML model. This is the case, because connectors in SysML are only allowed to connect ports with the same bit width. Changing the bit width of the *IO* port of the SysML block *ASWCSensor* to 16 bits, e.g. by a synchronization, without changing the bit width of the corresponding port of the block *HWSensor8Bit* leads to a violation of this property while the AUTOSAR model is still consistent.

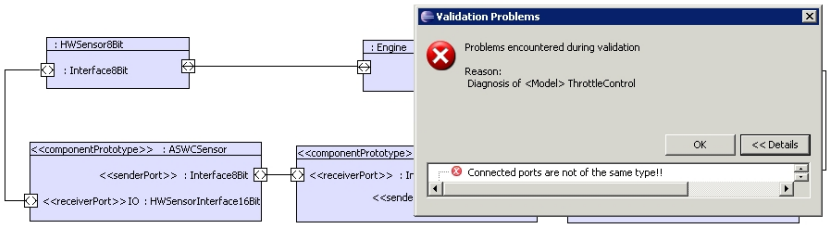


Fig. 16. Screenshot of the OCL validation dialog in TOPCASED

Ideally, the modeling tool should provide a mechanism to check models for syntactical and semantical correctness, not only to check models after synchronizations, but also to aid the user. TOPCASED, respectively EMF, provides such a validation mechanism. Validating the situation described above results in an error message like shown in Figure 16.

Currently, these constraint checks are not invoked automatically, but only on the user's request. The reason is that constraint evaluation in TOPCASED can take a very long time, because it always analyzes the whole model. Of course, it is desirable to check a model immediately after a synchronization for better usability.

8 Usage Scenarios

The described architecture supports several scenarios where, e.g., an initial AUTOSAR model is derived from an existing SysML model.

Additionally, the described architecture allows the synchronization of existing models by updating only changed model elements in the target model without overwriting the whole model each time changes occur. Such a synchronization can be executed in both directions. In the following, we describe different usage scenarios in which the shown architecture allows an enhanced development process using model transformation and synchronization techniques.

8.1 Transformation from SysML to AUTOSAR

After the SysML model has been constructed, it needs to be transformed into an AUTOSAR model to get from the system design to an initial model for the software design. Design decisions concerning the software defined in the SysML model have to be taken over to the AUTOSAR model. With the presented system such an initial AUTOSAR model can be automatically derived by a forward transformation. The automatic transformation is much faster than a manual transformation and there is less risk of introducing errors into the AUTOSAR model. A transformation in the other direction is also possible (backward transformation).

8.2 Repeated Forward Synchronization from SysML to AUTOSAR

After the AUTOSAR model has been derived from the SysML model, modifications can still be made to the SysML model. These modifications have to be transferred to the AUTOSAR model, too. While the AUTOSAR model already exists, a complete retransformation is unnecessary. Therefore, only the modifications are synchronized. Furthermore, the AUTOSAR model might also have been modified, e.g., by changing the type of the IO port of the ASWC *ASWC-Sensor* as described in Section 7. A complete retransformation would discard these modifications.

8.3 Backward Synchronization from AUTOSAR to SysML

However, modifications may also be made to the AUTOSAR model in order to adjust the structure during refinement of the software architecture, e.g., to reuse

an already existing component. Therefore, modifications also have to be propagated back to the SysML model. While many model transformation approaches only permit unidirectional transformations, our approach works bidirectionally and incrementally. Additional details in the SysML model are preserved which would otherwise be lost.

How such a propagation of changes using bidirectional transformation techniques supports the development process is demonstrated by the following scenario. When the type of the IO port of the ASWC *ASCWSensor* from Figure 4 is changed in the AUTOSAR model, the transformation system updates the corresponding SysML IO port shown in Figure 2 accordingly without overwriting the whole SysML model. When the SysML model is updated, the OCL constraint described in Section 3.1 is violated (see Figure 16) because the SysML connector is connected to ports, that have different types.

When elements have been added to the AUTOSAR model that are not relevant for the SysML model (e.g., on a more detailed abstraction level), these elements are ignored by the transformation system. This is the case, because no transformation rules have been defined for these elements.

8.4 Iterative and Flexible Processes

The usage scenarios outlined in Sections 8.1, 8.2 and 8.3 demonstrate that our approach can handle changes occurring in either model in any order. Therefore, the approach enables not only a strict sequential ordering, i.e. the SysML model is specified first and the AUTOSAR model is derived from it (Section 8.1). It also allows, that changes in the SysML model are propagated to an already existing AUTOSAR model (Section 8.2) and that necessary changes in the AUTOSAR model are also accordingly adjusted in the SysML model (see 8.3). Therefore, instead of a rigid sequential process, also iterative and more flexible processes can be supported. Later changes of the AUTOSAR model will be reflected back to the SysML model after a synchronization. Such changes in an AUTOSAR model can lead to the violation of constraints of the SysML model like described before.

9 Conclusion and Future Work

During the development of complex engineering solutions, several models are employed to capture the design decisions of different disciplines. We have presented an approach that supports synchronizing these models when they overlap with regard to the captured information. The solution enables that the interplay between the different development activities in different disciplines and the overarching system engineering can be kept consistent at minimal costs even though we do not forbid changes in the different models, which might impact each other or could lead to inconsistencies. The only limitation is that parallel changes in the different models are not supported. Although the underlying transformation

system can only synchronize two models, chains of transformations can be built to connect more than two models.[13]

We have further demonstrated that SysML models employed early on by system engineers and AUTOSAR models employed later on in the software development process can be kept consistent using our approach thanks to the use of model synchronization techniques and additional consistency rules. It has been further outlined that flexible usage scenarios, and in particular iterative development, become manageable when employing our approach.

As future work, we plan to further extend the coverage and also address other development artifacts than models. We also want to investigate how multiple models connected via model synchronization and consistency rules can be efficiently managed as a whole.

Acknowledgement

We would like to thank dSPACE GmbH for their support in developing the presented results and Oliver Niggemann, Joachim Stroop, Dirk Stichling and Petra Nawratil for their support in setting up and running the project.

References

1. Systems Modeling Language v. 1.1 (November 2008), http://www.sysml.org
2. Aldazabal, A., Baily, T., Nanclares, F., Sadovykh, A., Hein, C., Ritter, T.: Automated model driven development processes (2008)
3. AUTOSAR: UML Profile for AUTOSAR (January 2007), aUTOSAR GbR
4. Becker, S., Herold, S., Lohmann, S., Westfechtel, B.: A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. Software and Systems Modeling (SoSyM) 6(3), 287–315 (2007), http://dx.doi.org/10.1007/s10270-006-0045-5
5. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. In: GRaMoT 2008: Proceedings of the Third International Workshop on Graph and Model Transformations, pp. 25–32. ACM, New York (2008)
6. Borland Together Architect, http://www.borland.com/
7. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A.: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. International Journal on Software Tools for Technology Transfer (STTT) 6(3), 203–218 (2004), http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2004/STTT-BGN+04.pdf
8. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In: Richardson, J., Emmerich, W., Wile, D. (eds.) Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering, September 23, pp. 267–270. IEEE Press, Edinburgh (2002)

[13] An example is the chain SysML model=>AUTOSAR model=>SystemDesk model, although this chain of transformations was not originally intended.

9. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM System Journal 45(3) (July 2006)
10. Darmstadt, T.U.: Moflon (2007), http://www.moflon.org
11. Egyed, A.: Fixing Inconsistencies in UML Design Models. In: Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, MN, USA, pp. 292–301. IEEE Computer Society, Los Alamitos (May 2007)
12. Ermel, C., Rudolf, M., Taentzer, G.: The AGG Approach: Language and Environment. In: Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools, vol. 2. World Scientific Publishing, Singapore (1999)
13. Finkelstein, A.: A Foolish Consistency: Technical Challenges in Consistency Management. In: Ibrahim, M., Küng, J., Revell, N. (eds.) DEXA 2000. LNCS, vol. 1873, pp. 1–5. Springer, Heidelberg (2000)
14. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000), http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/1998/TAGT1998.pdf
15. France Telecom: SmartQVT, http://smartqvt.elibel.tm.fr/
16. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards final Standard. OMG, 250 First Avenue, Needham, MA 02494, USA (2003), http://www.omg.org/docs/ad/03-08-02.pdf
17. Giese, H., Hildebrandt, S., Neumann, S.: Towards Integrating SysML and AUTOSAR Modeling via Bidirectional Model Synchronization. In: 5th Workshop on Model-Based Development of Embedded Systems (MBEES) (2009)
18. Giese, H., Hildebrandt, S., Seibel, A.: Improved Flexibility and Scalability by Interpreting Story Diagrams. In: Magaria, T., Padberg, J., Taentzer, G. (eds.) Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009) (2009)
19. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1) (February 1, 2009), http://www.springerlink.com/content/j716245824112n27/
20. Guerra, E., de Lara, J.: Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 54–69. Springer, Heidelberg (2004)
21. ikv++ technologies ag: medini QVT (2007), http://www.ikv.de
22. Johnson, T., Paredis, C., Burkhart, R.: Integrating Models and Simulations of Continuous Dynamics into SysML (2008)
23. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
24. Kindler, E., Rubin, V., Wagner, R.: An Adaptable TGG Interpreter for In-Memory Model Transformation. In: Proc. of the Fujaba Days 2004, Darmstadt, Germany, pp. 35–38 (September 2004)
25. Marschall, F., Braun, P.: Model Transformations for the MDA with BOTL. In: Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications. Technical Report TR-CTIT-03-27, Univeristy of Twente (June 2003)
26. Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification (January 2006), document ptc/06-11-01

27. OMG, O.M.G.: Object Constraint Language (May 2006), http://www.omg.org/spec/OCL/2.0/PDF
28. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In: Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools, vol. 2, pp. 487–550. World Scientific Publishing Co., Inc., River Edge (1999)
29. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903. Springer, Heidelberg (1995)
30. Stürmer, I., Kreuz, I., Schäfer, W., Schürr, A.: Enhanced Simulink Stateflow Model Transformation: The MATE Approach. In: Proc. of MathWorks Automotive Conference (MAC 2007). Dearborn (MI), USA (2007)
31. Tata Consultancy Services: ModelMorf (2007), http://www.tcs-trddc.com/ModelMorf/index.htm
32. University of Paderborn. Fujaba Tool Suite, Germany, http://www.fujaba.de/
33. Varró, D., Varró, G., Pataricza, A.: Designing the Automatic Transformation of Visual Languages. Science of Computer Programming 44(2), 205–227 (2002)
34. Vizhanyo, A., Agrawal, A., Shi, F.: Towards Generation of Efficient Transformations. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 298–316. Springer, Heidelberg (2004)
35. Windpassinger, H.: Modellierungssprache für die Kfz-Software Entwicklung. Elektronik Praxis (2007), http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/analyseentwurf/articles/95528/

Multi-view Modeling to Support Embedded Systems Engineering in SysML

Aditya A. Shah, Aleksandr A. Kerzhner,
Dirk Schaefer, and Christiaan J.J. Paredis

Systems Realization Laboratory,
G. W. Woodruff School of Mechanical Engineering,
Georgia Institute of Technology, Atlanta, GA, USA
aditya.shah@gatech.edu, alek.kerzhner@gatech.edu,
dirk.schaefer@me.gatech.edu, chris.paredis@me.gatech.edu

Abstract. Embedded systems engineering problems often involve many domains, each with their own experts and tools. To help these experts with analysis and decision making in their domain, it is desirable to present them with a view of the system that is tailored to their particular task. In this paper, a model integration framework is demonstrated to address issues associated with multi-view modeling. The Systems Modeling Language (OMG SysML™) is used as a general language to represent a common model for the system as well as the dependencies between the different domain-specific tools and languages. To maintain consistency between these domain-specific views, model transformations are defined that map the interdependent constructs to and from a common SysML model. The approach is illustrated by means of a mechatronic design problem involving views in multiple domain-specific tools, namely EPLAN Fluid™ (to create production ready layouts) and Modelica® (for dynamic system analysis).

1 Introduction

The design of embedded systems is becoming increasingly complex because of its dependence on a combination of multiple disciplines, such as mechanical, electrical, electronics and software engineering. Moreover, since such systems are designed in collaborative distributed environments, they need to meet the objectives of a variety of stakeholders. To solve these multi-disciplinary embedded systems problems, it is essential to be able to synthesize and analyze different system architectures efficiently and effectively. This requires the ability to describe a system from different viewpoints, such as: structural, behavioral, requirements, different disciplinary domains, or different levels of detail, fidelity and abstraction. Therefore, multi-view modeling, in which a system is described from multiple points of view, is gaining in popularity.

A key challenge in dealing with multiple views of a system is that of consistency. This is conceptually illustrated in Fig. 1. Information about the product or system (e.g., the requirements, the physical structure, the behavior, or the

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 580–601, 2010.

manufacturing-process specification) is represented by black dots; the dependencies between the information objects is shown as boxes labeled 'D'. Examples of such dependencies include an assembly-part relationship, the dependence between the mechanical length of a wire and its electrical resistance, or the dependence between a circuit description and the corresponding analysis model. During the product development process, new information is generated by the stakeholders using a variety of domain-specific tools: synthesis or authoring tools (e.g., CAD tool or drawing tool for hydraulic schematics) as well as analysis tools (e.g., dynamic simulation tool, or reliability analysis tool).

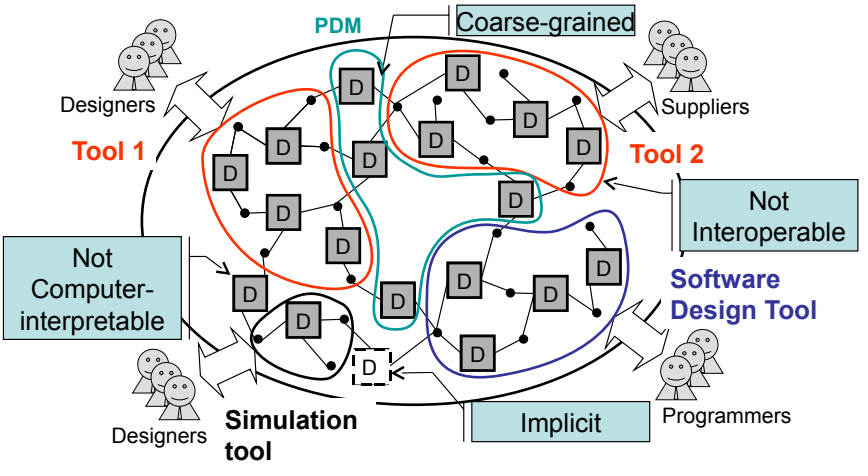


Fig. 1. An example of an embedded system problem with multiple subsystem models in different domains with interdependencies between the models. Dependencies (D) exist between certain parts of different domains but not across the complete domain. A method that can identify these dependencies as well as allow for the addition of knowledge between models is required, such as the method presented in this paper.

Ideally, the information and dependencies defined within such tools would be linked globally across tool boundaries. But because individual tools are not aware of the information in other tools, the dependencies across tool boundaries often remain unaccounted for. Currently, at best, such dependencies are maintained at a coarse-grained level in a Product Data Management (PDM) system. For instance, one may keep track of which finite element model is related to a certain CAD file. But such coarse-grained dependencies do not identify at a fine-grained level which parameters in the finite element model depend on which CAD parameters. Even worse, many of the dependencies are modeled only informally (e.g., in a text document) or not at all (e.g., they are unknown or exist only in the brain of a stakeholder).

Still, not all information needs to be shared among all tools. For instance, the detailed description of the embedded software can be maintained in a software engineering tool but does not need to be shared with a mechanical CAD (MCAD) tool.

A fine-grained approach is needed that can identify information that is or is not common between multiple views is needed. For instance, while component and connectivity information from electrical or hydraulic schematics are important for simulation-related views, diagram layout and placement details may not be relevant in this particular context. Moreover, additional knowledge is often required when integrating between views, such as specifying control and simulation parameters when creating a simulation model from a structural model. Therefore, in such cases where models cannot be completely converted using standard interchange formats, maintaining consistency is usually left to the users resulting in significant non-value-added efforts and potential for error.

In order to avoid such costly and error-prone processes, the approach advocated in Model-Based Systems Engineering (MBSE) is to model both the various views and their dependencies formally in terms of a common system model. However, even when using such a common system model, the modeling formalism must explicitly allow for information to be shared between different views. In this paper, the focus is to demonstrate a framework in which different domain-specific views can automatically be generated from a common system model. We explore and develop a foundation for supporting multiple views in the Systems Modeling Language developed by the Object Management Group (OMG SysML^{TM}) [1]. Through domain specific modeling and model transformations, multiple domains and their relations can be described. Moreover, the framework presented complies with industry standards such as Meta-Object Facility (MOF) [2], Model Driven Architecture (MDA) [3], and SysML, thereby facilitating its integration with a variety of existing tools and standard frameworks.

In the next section, related work and the motivation for the use of SysML is discussed. In Section 3, we present a method for multi-view modeling in the context of an electro-hydraulic embedded systems example. Finally, Section 4 summarizes the work presented and discusses the challenges that exist with regard to this framework and multi-view modeling frameworks in general.

2 Related Work

Embedded systems traditionally involved only the software domain. However, the increasing use of software as a replacement for hardware has resulted in embedded systems becoming a part of many complex systems in practice. As a result, embedded systems design spans multiple domains including the software to control these systems. These domains contain different information maintained in multiple views for each of the various subsystems. Therefore, the requirements for maintaining consistency between multiple views are different than those for maintaining interoperability between models.

Interoperability involves managing similar information across different formats. For instance, standard file formats such as XML [4] and STEP [5,6], are commonly used for interoperability in the software and engineering domains. On the other hand, multi-view modeling involves managing *different* information across different tools and domains. In such cases, tools usually do not conform to common file formats and if they do then there are limitations such as data loss issues in STEP [7] and readability issues with XML and XMI. Moreover, such transformations are usually carried out in batch mode and hence maintaining consistency and updating various views can become cumbersome [8].

Since different views involve different knowledge, such as synthesis or analysis knowledge, only certain aspects of each view are related. Therefore additional knowledge is required to generate one view from another, such as an analysis view from a structural view. Model transformations are suited for this, in which a source model is converted into a target model based on specified rules. Czarnecki *et al.* [9] discuss the classification and comparison of different model transformation based approaches, such as direct manipulation and graph transformations. In direct manipulation, general purpose programming languages such as Java or C are used directly to define transformations. As a result, it can be cumbersome to use, understand and maintain due to the lack of high-level abstractions available. Graph transformations, however, can be defined in a declarative and visual manner [10], allowing complex rules to be expressed in an intuitive fashion [11]. Another benefit of graph transformations is that they can be executed in continuous or incremental modes. Therefore, transformations can be performed interactively with the user which makes managing consistency between various views easier [8]. In addition, Computer Aided Software Engineering (CASE) platforms such as Fujaba [12] and MOFLON [13] provide the capability to define graph transformations and automatically generate corresponding executable code for performing the model transformation in a language such as Java. Therefore, in the research described in this paper, a graph transformation approach is adopted for the integration of multiple views of a system.

In addition to multi-view integration, a mechanism for systems level design is required for embedded systems that are large and highly complex. To overcome these problems, the use of a general-purpose modeling language such as SysML has been proposed as an alternative to UML [14,15,16]. The use of SysML as a unifying language for systems design is discussed in the next section, along with related work that is being carried out with SysML.

2.1 SysML as a Common Modeling Language for Multi-view Modeling

As discussed in the previous section, due to the widespread use of embedded systems in contemporary products and systems, there is an increasing requirement for multi-view modeling methods that can incorporate multiple disciplines as well as integrate between their associated views. Multi-view modeling also involves describing a particular system in multiple ways, such as structural, behavioral, and requirements views.

UML has been widely used in software engineering to model multiple views of a software system. However, since embedded systems are used in multiple disciplines of a complex system, embedded systems can no longer be designed from a purely software modeling point of view. Consequently, to overcome this limitation of UML when dealing with domains other than software, we advocate the use of SysML – a general Systems Modeling Language – that provides certain key extensions of UML to enable modeling of multidisciplinary systems, including software. SysML extends UML in the form of a SysML profile. Some key features include the introduction of requirements modeling, extension of `Class` in UML to `Block` in SysML to model system structure, constraint blocks to support engineering analyses, and the use of ports and flows to support flow of energy, information or matter between elements. More information regarding the specification and features of SysML can be found in [1,17].

Vanderperren and Dehaene [15] highlight the advantages of using SysML for requirements modeling in embedded systems (System on Chip) as well as to improve current UML-based development techniques. In addition to these new features, SysML and UML provide a mechanism to customize the general semantics of SysML through profiles. This is an important part of the multi-view modeling method presented in this paper, and is discussed in more detail in Section 3.3 on Domain Specific Modeling in SysML.

In order to support multiple complementary views of a system we propose to use a common model in SysML that consists of a combination of relevant knowledge from the domain-specific views at a high level of abstraction. In addition, we use profiles to enable the domain-specific description of different disciplines and languages in SysML, as shown in Fig. 2. When compared to Fig. 1, we see that the dependencies are not directly between the multiple disciplines; rather, they exist indirectly through a common model for the problem under consideration. Consequently, each domain-specific model is a subset of the common model and is therefore a view of the system. Since each domain-specific view is at a more detailed level of abstraction compared to the common model, only certain aspects are required to be mapped between the common model and the corresponding view. This is achieved by abstracting the domain-specific semantics mapped into SysML through the use of profiles. As a result, only those parts of domains that are related to each other are integrated instead of translating a complete model from one domain to another. This can allow the designer to concentrate on specifying the system without worrying about the semantics and features of specific tools. Moreover, since the different domain-specific views are complementary to each other, maintaining a common high-level representation in SysML provides designers with a consistent big-picture view of the problem being solved.

The framework for multi-view modeling presented in this paper involves describing domains through SysML profiles and metamodels, and defining mappings between them through model transformations. This method is discussed in the following section through an example of an electro-hydraulic system that

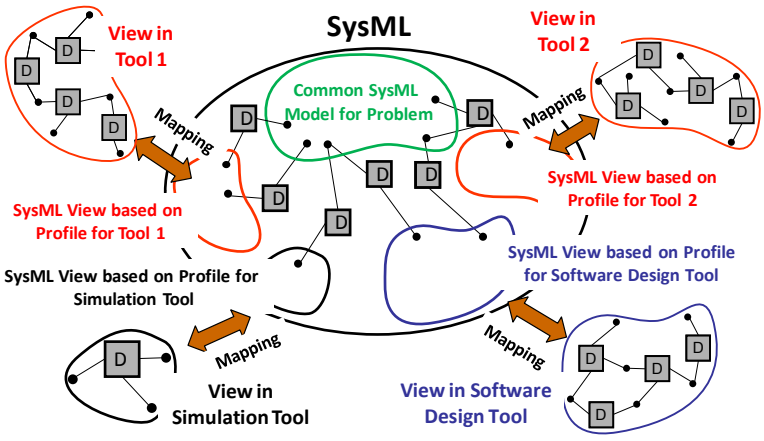


Fig. 2. Use of SysML and mappings as a common modeling language in which dependencies between different domains are captured through the use of a common view of the system in SysML.

has multiple views - a schematic layout view and a continuous simulation view in which the control system is modeled.

3 Multi-View Modeling Using SysML Profiles and Model Transformations

Our general method for multi-view modeling involves the following steps, which will be described in the context of an example problem of an electro-hydraulic log splitter.

1. Formal definition of the domains involved in the system through metamodels.
2. Customization of SysML through profiles to enable domain specific modeling.
3. Model transformations to generate domain-specific views from SysML

3.1 Example Problem Description

To demonstrate the implementation of the above steps, an example is used that can be modeled in different views such as structural, control and simulation. The example used for the remainder of the paper is of an electro-hydraulic log splitter (Fig. 3), which is used to divide cylindrical logs into two or more pieces. A traditional log splitter includes a hydraulic circuit in which a valve is manually operated to actuate a cylinder into a piece of wood. Instead of manual valve operation, we consider an embedded control system to operate the valve, resulting in improved functionality but also additional subsystems. These different subsystems are usually represented in multiple domain-specific

tools such as EPLAN Fluid™ [18] to capture the hydraulic schematics and Modelica® [19] for modeling the control system and dynamic behavior. In order to support these complementary views, a common model in SysML is created in order to represent the entire system at a high level of abstraction from which EPLAN and Modelica specific views can be created with the relevant subset of information (see Fig. 4). This is similar to the approach of domain-spanning specification by Gausemeier *et al.* [20] and Product View Federations by Bajaj *et al.* [21]. Thus, the integration between EPLAN and Modelica is done indirectly through a common model from which views can be generated by using SysML profiles and model transformations.

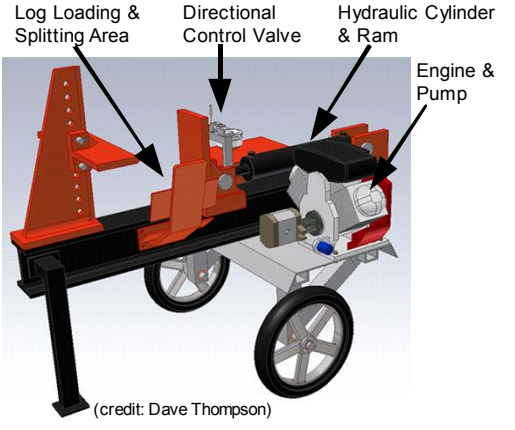


Fig. 3. An electro-hydraulic log splitter. The production-related hydraulic layout is done using EPLAN Fluid, while a dynamic simulation involving the hydraulics and its control system is done in Modelica. A common view for both will be represented in SysML, from which views for EPLAN and Modelica will be created.

From the common model two views are created: a structural hydraulic schematic in EPLAN (Fig. 5) and a dynamic simulation of the system behavior in Modelica (Fig. 6). These two views each contain a subset of the knowledge captured in the common model. For instance, both tool-specific views contain the components and connection information. However, the EPLAN view does not contain controller information while the Modelica view does not contain information related to geometric layouts. A designer using EPLAN or Modelica may add/remove components and this is where a common model is useful in maintaining the relevant changes on both sides. The process of maintaining consistency can be similar to the process used by Gausemeier *et al.* [20], in which problem-specific rules and intervention by experts can be used to manage update and propagation between views. However, since each of the views are in separate tools, this paper presents an initial effort for creating multiple views that are linked together by a common model in a common representation. Since the update and propagation of changes involve custom problem-specific rules, it

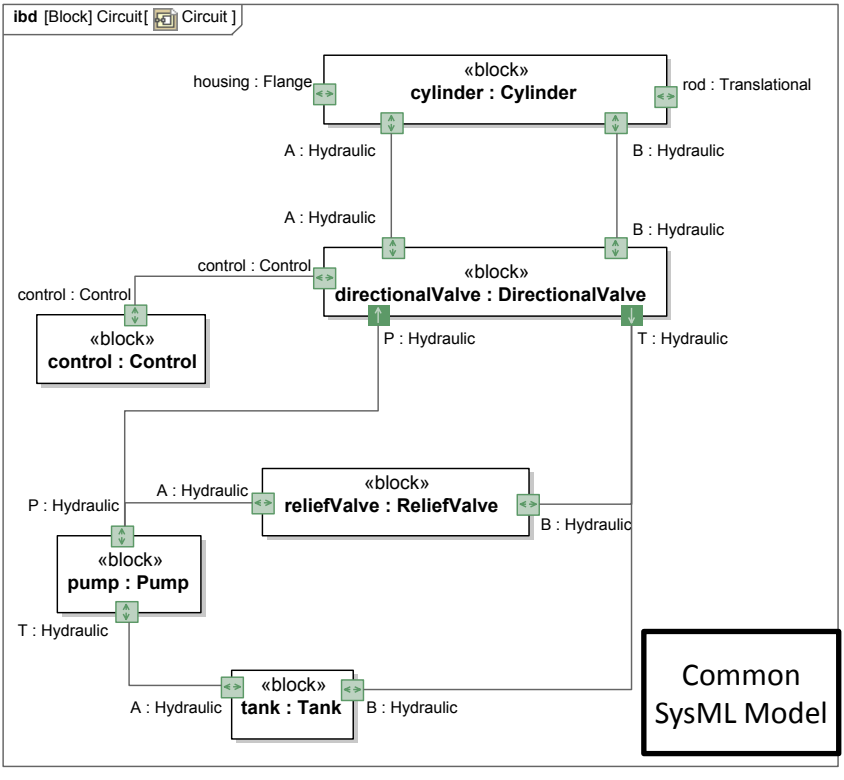


Fig. 4. A common model representation of a log splitter in SysML. The EPLAN view does not need control system models while the Modelica view does not require layout information. However, both are represented in SysML as black-boxes which can further be developed in the respective tools and languages.

is not considered in this paper; however such rules can be included within the method presented.

3.2 Formal Capture of Domain Knowledge Using Metamodels

When dealing with domain-specific views of a system, a formal representation of each view provides a more structured and object-oriented representation to interpret. This is normally done through internal data-models that are customized for a particular domain-specific software package. The structure of these internal data-models is often captured only in the source code of the software package. Therefore, to integrate multiple views, the first step is to represent each domain or tool formally in a common format through the use of metamodels. A metamodel is a "model used to model modeling itself" [2]. In other words, a metamodel for a domain is a model that specifies the possible concepts that can

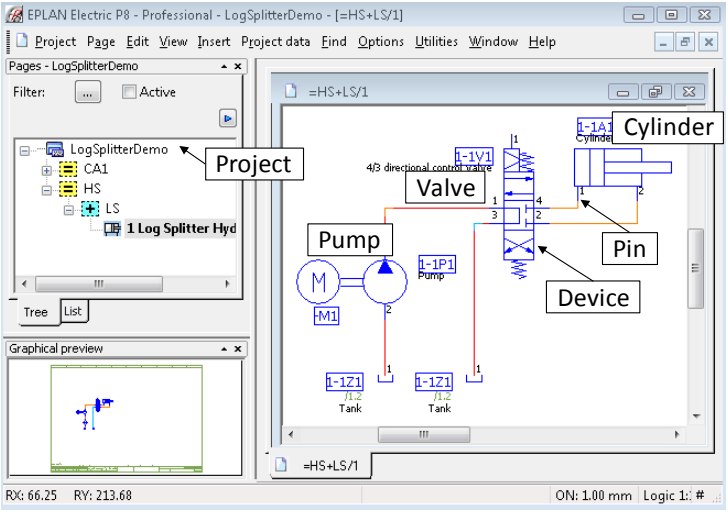


Fig. 5. A schematic for an Electro-hydraulic Log Splitter represented in EPLAN. From the common model in SysML, an EPLAN view is created in which a designer adds knowledge related to the layout of the system.

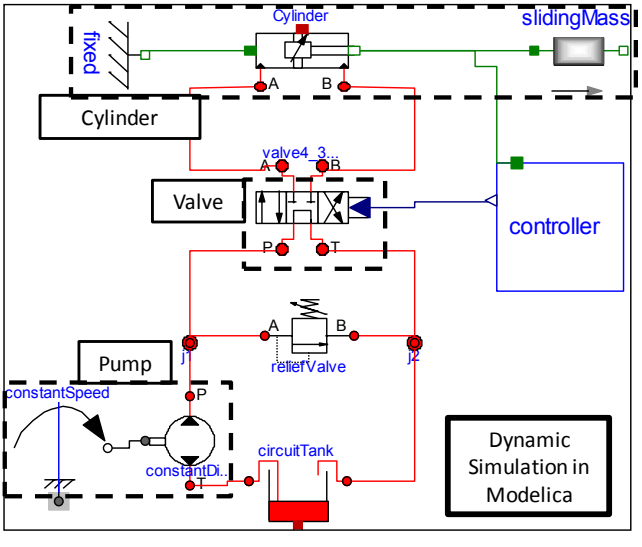


Fig. 6. A dynamic simulation model for the log splitter in Modelica. Similar to the EPLAN view, the Modelica view is created from the common model in SysML. The designer adds knowledge related to the control system and then performs an analysis of the system.

exist within that domain. Each view of a system is therefore an instance of a metamodel defined for its domain.

To support model and metamodel driven systems, OMG established the Meta Object Facility (MOF) [2] standard. MOF provides a methodology and framework for "defining, manipulating, and integrating meta-data and data in a platform independent manner" [2,22]. The approach we have taken is to specify explicitly the metamodel reconstructed from the API of the tool with which interoperability is desired [23]. This is a conversion of the implicit metamodel (i.e., the data structures used internally to the tool and only made visible through its API targeted for general purpose programming languages like C++) into a formal and explicit metamodel compliant with the MOF standard. The MOF compliant meta-CASE tool MOFLON [13] is used to define the metamodels for all of the domains involved.

As an example, the abstract syntax of EPLAN Electric P8, which is accessible through its API, is explicitly defined in terms of a metamodel in MOFLON, of which a small part is shown in Fig. 7. EPLAN constructs, such as `Project`, `Page`, `Function`, `Connection` are specified as classes while the relationships between the constructs are specified as associations (`ProjectHasPages`, `PageHasDevices`, etc.).

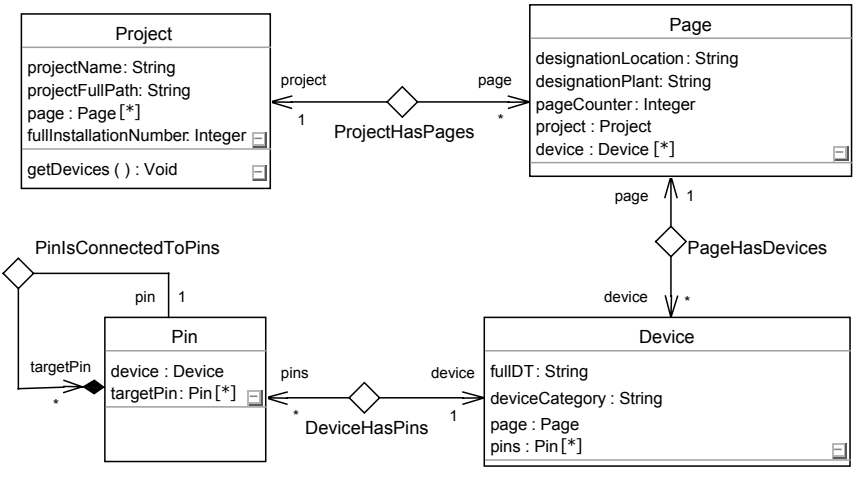


Fig. 7. A part of a metamodel for EPLAN defined in MOFLON.

A similar approach is taken to define a metamodel for Modelica. In contrast to EPLAN, the abstract syntax of Modelica is contained within another metamodeling language known as Meta-Modelica [24]. Therefore, using the Modelica metamodel as a guide, a corresponding MOF metamodel is defined, as shown in Fig. 8.

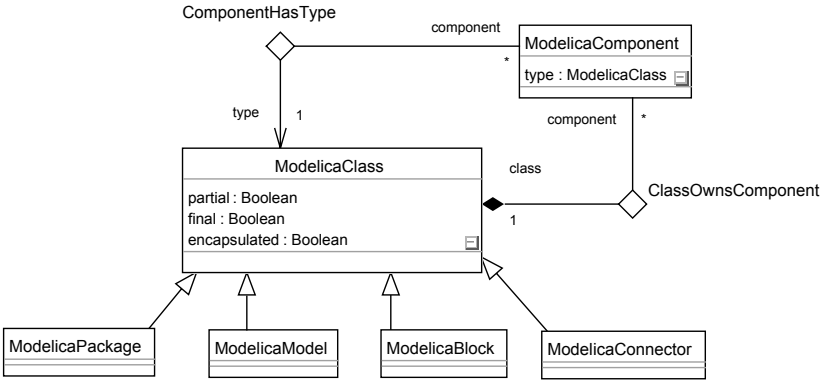


Fig. 8. Part of metamodel for Modelica defined in MOFLON.

In both cases, only those portions of the language that are needed for defining the required transformations are captured within each MOF metamodel. The metamodels can be modified to incorporate more features of the language as well as ensure compatibility with new versions. After defining each domain by an explicit MOF metamodel, it is necessary to customize SysML to enable transformation between an instance of the metamodel and a corresponding instance in SysML. This is done through the use of SysML profiles and model libraries, which is the next step in our framework for multi-view modeling.

3.3 Domain Specific Modeling in SysML

In order for SysML to be used as a bridge between multiple domains, the necessary semantics from each of the domains must be included within SysML. However, since SysML is a general purpose modeling language, it lacks the detailed, formal semantics needed to represent specific views in an executable format [25]. For instance, a model that is described using generalized constructs such as SysML Blocks is just a visual representation. It cannot be executed, unlike a Modelica or Simulink model that can be simulated or an EPLAN model that can be used to create assembly schematics or a Bill of Materials. In addition, it can be cumbersome for domain experts to create models in SysML, thereby limiting the acceptance of general SysML for specific domains. To overcome this limitation, SysML can be customized to define semantics specific to a domain. Indeed, most UML / SysML tools allow for a broad range of customizations through the use of Domain Specific Languages (DSLs) and corresponding modifications to the graphical user interface.

Among the various approaches available to define DSLs in SysML [26], profiles are preferred since they do not modify the underlying SysML metamodel so that SysML tool support is retained. A portion of a SysML profile created for EPLAN is shown in Fig. 9. The profile is constructed as per the MOF metamodel (Fig. 7).

Language constructs specific to the EPLAN (e.g., `Page` and `Device`) are defined as stereotypes that extend existing SysML and UML constructs, such as the `Block` metaclass of SysML.

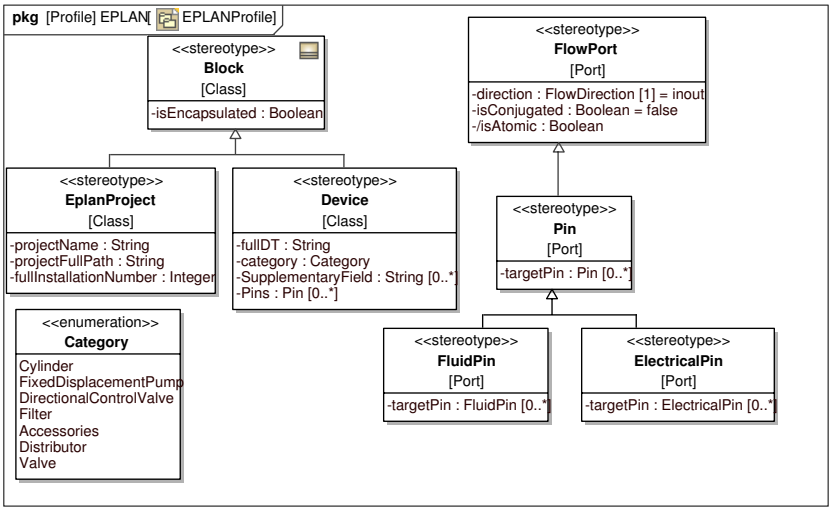


Fig. 9. A Profile for EPLAN in SysML. Stereotypes such as *EplanProject*, *Device*, *Pin* represent constructs used in EPLAN.

A profile is also defined to represent Modelica simulation models in SysML. Rather than creating a stereotype for every language construct of Modelica, only the subset of constructs necessary for capturing the model interfaces is modeled. Such "black box" models refer to externally defined Modelica models for their internal details. The SysML profile includes the full path to the library and the specific model within it [27], as is shown in Fig. 10.

Using these profiles, one can represent any EPLAN or Modelica model within SysML, and then establish the dependencies between them using the common model. To facilitate this process, it would be convenient to map entire libraries of models to their corresponding SysML-profile versions so that they can be referenced by systems engineers working in a SysML tool. At present, these SysML model libraries are created manually but the process could be automated by using a tool's internal API (e.g., as for EPLAN) or by parsing through a model library file (e.g., as for Modelica). By storing models that reference preexisting tool-specific models, the designer's need to know detailed syntax for each tool is reduced. Consequently, a designer can focus more on the problem to be solved instead of having to learn the syntax of each tool.

In conclusion, the combination of profiles and model libraries with metamodels provides the framework in which model transformations can be applied to integrate multiple views. This is discussed further in the next section.

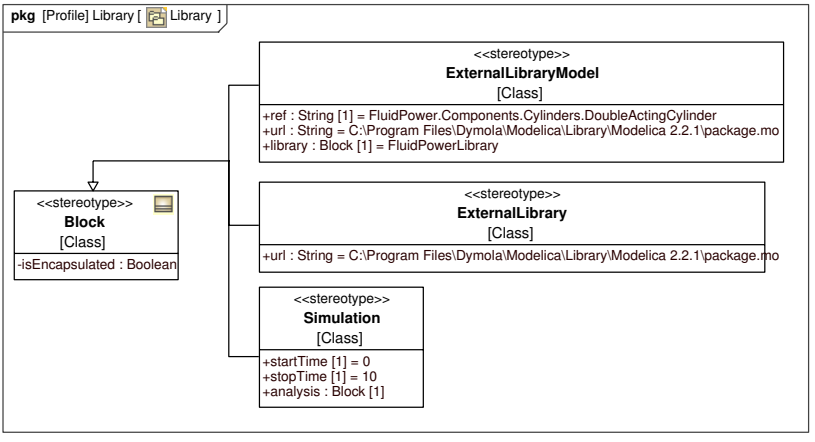


Fig. 10. A Library profile is used to capture model library elements. Stereotypes such as *ExternalLibrary* and *ExternalLibraryModel* are used to specify the reference locations that point to the actual tool-specific models.

3.4 Model Transformations to Generate Domain-Specific Views

Once all of the domains are represented by MOF metamodels and SysML profiles, transformation rules can be defined both create domain-specific views in SysML from the common model and create domain-specific tool representations of the views. Model transformations are commonly used to convert a model in one domain (the source model) to a model in another domain (the target model), as is shown in Fig. 11. When creating domain-specific SysML views, the source model is the common SysML model and the target is a domain-specific view in SysML. For generating domain-specific tool representations, the source model becomes the domain-specific SysML view. Since both metamodels and SysML profiles can be described in terms of graphs [10], graph transformations can be used to integrate between the different views and models. In this case, the domain semantics (metamodel and profile objects) represent the nodes, and associations represent the edges.

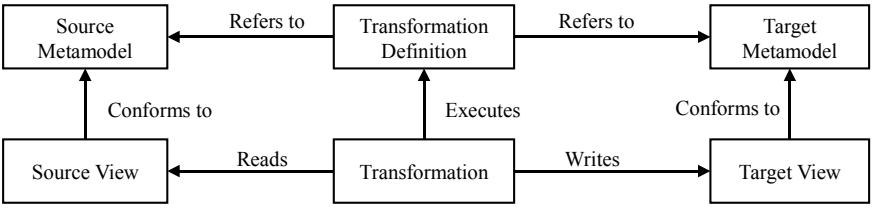


Fig. 11. Process of model transformation from Source to Target model (Czarnecki *et al.* [28]

Also, many common transformation-based approaches can be used including OMG's Queries / Views / Transformations (QVT) [29] and Triple Graph Grammars (TGG) [30] because both the source and target models are instances of formally defined metamodels. In our approach, story diagrams [31] are used to visually define rules for the transformations between different views. These rules are TGG-like in that they use a correspondence graph that captures the relationships between source and target model. Just as in TGGs, the correspondence graph is defined by a correspondence metamodel [22,30]. For instance, in Fig. 12 a correspondence metamodel is used to link objects of a SysML view to objects in an EPLAN tool specific view. The object `DeviceBlock2Device` links a `Block` object in SysML to a `Device` object in the Electrical CAD (ECAD) domain, just as the stereotype `Device` is a generalization of the `Block` metaclass in the SysML profile. Story diagrams are used because they allow the inclusion of API specific code.

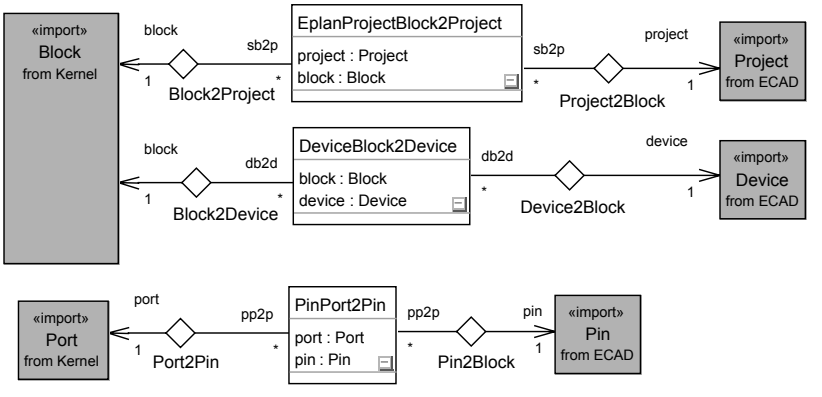


Fig. 12. Correspondence metamodel between SysML and EPLAN. It is used to maintain relation between elements of different views. A *Block* in SysML is related to a MOF object of class *Project* through the correspondence MOF object of class *EplanProjectBlock2Project*.

Unlike TGGs, the story diagrams presented are not specifications of bidirectional transformation rules and therefore rules are required in each direction. For instance, rules are defined between the common model and the structural view in SysML, between the structural view in SysML and EPLAN, between the common model and the Modelica view in SysML, and finally between the Modelica view in SysML and the executable simulation in Modelica. The transformations are defined using a Meta-CASE (Computer Aided Software Engineering) tool called MOFLON [13], which simplifies the overall process with its modeling and automated code generating capabilities.

A disadvantage of story diagrams is the need to specify the order of execution within the specification. Consequently, custom story diagrams are needed to implement problem-specific consistency management. Also, custom diagrams

would be needed to do incremental model construction. This is in contrast to TGG approaches in which consistency management rules can automatically be derived from the transformation specification such as in work by Königs and Schür [32]. TGGs can also be modified for incremental model construction as shown in work by Griese and Wanger [33].

In Fig. 13, a story diagram is shown that takes as input a SysML block (stereotyped with `EplanProject`) that contains path information for an EPLAN project file (`LogSplitter.elk`) and creates a `Project` object as per the EPLAN metamodel as well as updates the EPLAN project with information such as `ProjectName`. Other transformations (not shown) are then executed in order to fill in the EPLAN project from the SysML view. For instance, a transformation is defined that finds all the SysML blocks stereotyped as `Device` and creates devices in the EPLAN project along with the corresponding connections between them. Through such transformations, an input structural view in SySML as shown in Fig. 14 is converted into the corresponding EPLAN view as in Fig. 5. Not only can model transformations be used to create the entire model as shown here, they can also be defined to only propagate changes using correspondence information to find and update the appropriate model elements.

Both the transformations from SysML to EPLAN and SysML to Modelica establish dependencies across different modeling environments, as illustrated in Fig. 2. In addition, it is necessary to capture the dependencies between different views and the common model within the SysML environment. For examples, the EPLAN profile can be used to describe the structure of electrical or hydraulic circuits within SysML. This structural view is related to a corresponding analysis view that can be represented using the Modelica profile and linked using the common model. It is often possible to define the dependencies between different systems views parametrically using SysML parametric diagrams, and this approach can be augmented with model transformations to create dependencies. Parametric dependencies alone are sufficient when the structure of the views remains the same and only the parameter values need to be modified or updated. However, in this case, it is quite common to change the topology of the different views also. A parametric mapping would have to be re-established each time the topology of the structure is modified, a cumbersome and error-prone process.

The dependencies between the common model and domain-specific views have been modeled using model transformations. These model transformations embody the generic knowledge needed to create the domain-specific views from the knowledge available in the common model. These model transformations need to be complemented by domain-specific knowledge stored in mapping models, called Multi-Aspect Component Models (MAsCoMs) by Jobe *et al.* [34]. For instance, the EPLAN structural view contains a pump component, which refers only to the structural information about the pump: the type of pump, the key sizing parameters such as displacement and maximum pressure, and a product ID which are all present in the common model. With this information, a corresponding MAsCoM can be identified in a MAsCoM library, which in turn allows the model transformation to instantiate the appropriate EPLAN component model and

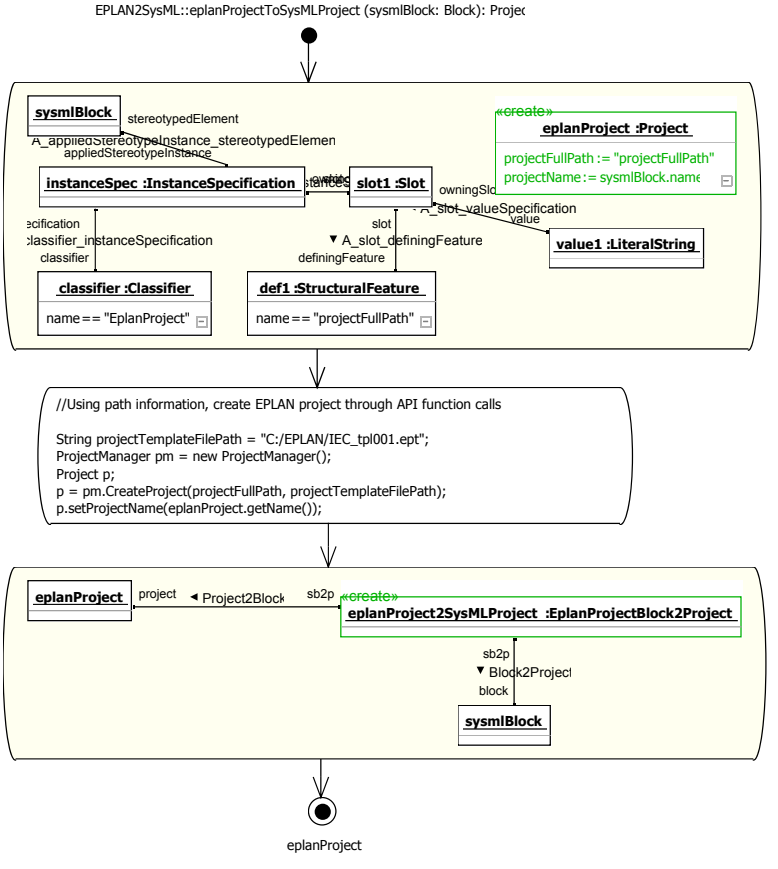


Fig. 13. A story diagram that creates an EPLAN project file with information from a SysML model. The input (*sysmlBlock*) contains project location and through EPLAN's API, project information is accessed and is added to the SysML input block.

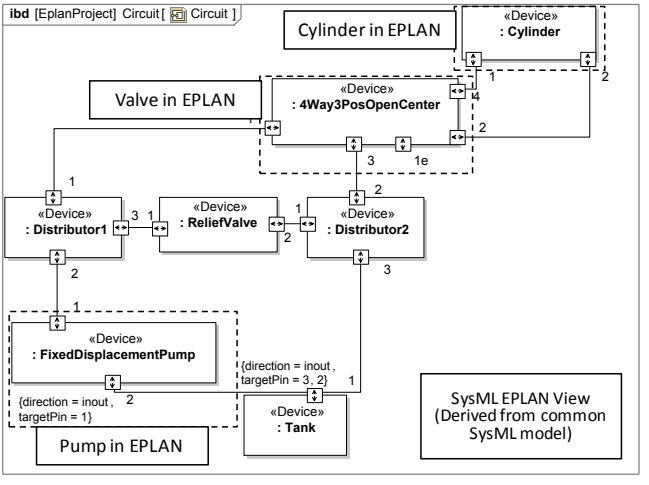


Fig. 14. Model transformations are executed to convert this view in SysML to EPLAN. The SysML model is in the form of an Internal Block Diagram, within the block that corresponds to the EPLAN project.

establish the parametric dependencies. (See Fig. 15) Refer to [34] for the details of this mapping approach. In this same way, transformations are executed to generate a Modelica view in SysML from the corresponding common model in SysML, as shown in Fig. 16.

Finally, once the Modelica model in SysML is generated, it is converted to executable Modelica code through model transformations defined on the Modelica metamodel (Fig. 8) and subsequent conversion to textual code through a tool integrator. The textual Modelica model can be executed in any Modelica solver such as the OpenModelica Compiler [35] or Dymola [36]. Figure 6 shows the generated Modelica model within Dymola.

The model transformations discussed above are defined in MOFLON, which automatically generates JMI (Java Metadata Interface) code that implements the transformations in Java. This JMI code is combined with a JMI-compliant SysML tool, such as Magic Draw [37], in the form of a plugin. To handle the incompatibility of programming languages of different tools (e.g., EPLAN is based on .NET while Magic Draw and MOFLON are based on Java), interoperability software may be necessary to bridge the gap. In the implementation of this example, JnBridge Pro [38] has been used to access the EPLAN API from within a Java-based model transformation framework.

In conclusion, through the successive execution of graph transformations, it is possible to generate several domain-specific views and domain-specific tool representations from a common model in SysML, thereby integrating tools that are otherwise incompatible with each other and encoding knowledge that would otherwise require manual regeneration of views each time the system topology changes.

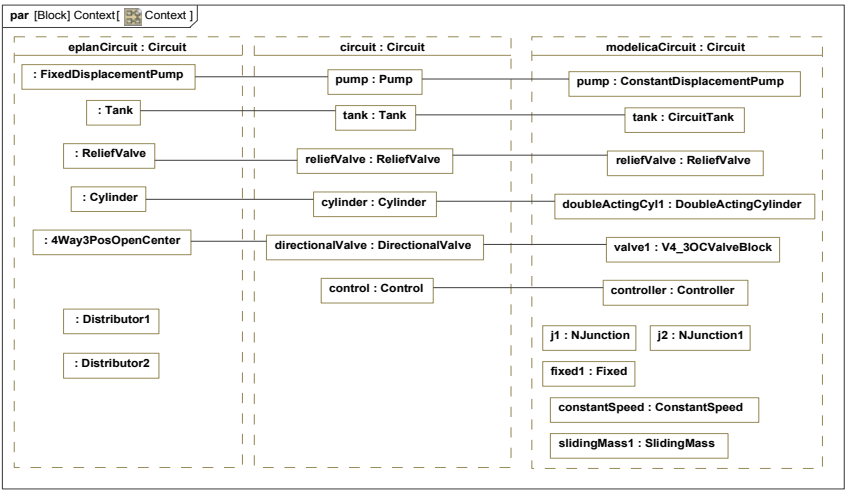


Fig. 15. Dependencies between the common model and domain-specific views in SysML.

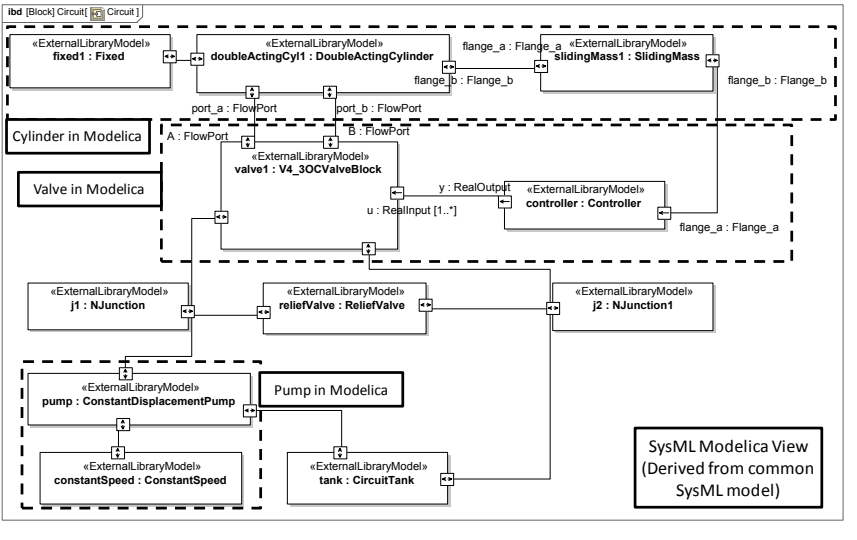


Fig. 16. Model transformations are executed to convert the common model in SysML to a Modelica model in SysML. The SysML model is in the form of an Internal Block Diagram.

4 Discussion and Closure

In this paper, the use of SysML profiles and model transformations for multi-view modeling of embedded systems is discussed. Embedded systems are becoming more prevalent, resulting in the increasing use of software throughout a variety of systems [14]. Although considerable research has been done for the automation of software development through UML, little effort has been focused on the model-based engineering of entire systems, including both software and hardware. To support the development of embedded systems, no single language such as UML or SysML can effectively capture all of the needed information. For instance, SysML is well-suited for defining the high-level relationships that exist between requirements, structure, and behavior. It is not suited for specifying schematics for assembly (EPLAN's function) or simulating control systems and dynamic system behavior (Modelica's function). Therefore, the method presented in this paper integrates multiple views that involve *different* knowledge (structural or analysis knowledge), thereby leveraging the capabilities of different domain-specific tools. As a result, one can synergistically combine the strengths of multiple modeling languages and tools. In SysML, the designer can trace requirements to desired behavior or allocate functions to structural components; these capabilities can now be combined with the modeling views created in tools that are specifically tailored for authoring or analyzing systems in formalisms with which domain specialists are most familiar.

However, as shown in the previous section, there are a number of challenges still to be addressed when creating a general framework for multi-view modeling. A major research question involves determining an effective way maintain consistency in a problem independent fashion. This is in contrast to the current approach of using problem specific update and propagation rules [20]. But it is not clear which approach is most suitable for maintaining and managing consistency between the different views. Ideally, a general framework must also take into account the work flow process, to allow consistency to be evaluated and reestablished periodically but not enforced all the time. This process is likely to require human involvement to resolve inconsistencies when multiple views cannot be reconciled without ambiguity.

These challenges are applicable to any framework and we believe that the model-based framework demonstrated in this paper is sufficiently flexible to accommodate extensions that will address these challenges. Moreover, additional design capabilities such as the design synthesis algorithms by Kerzhner and Paredis [27] and Helms *et al.* [39] can benefit from the model-based framework defined in the paper. In conclusion, the framework presented in this paper is an approach that we believe is not restricted to any particular domain and can serve as a step towards the unification of the various domains involved in embedded systems design.

Acknowledgments. This work has been funded by Deere & Company. Additional support was provided by the ERC for Compact and Efficient Fluid Power, supported by the National Science Foundation under Grant No. EEC-0540834.

The authors would like to thank Roger Burkhart, Sanford Friedenthal, Leon McGinnis, and Russell Peak for the discussions that helped crystallize the ideas presented in this paper. No Magic Inc. provided access to its MagicDraw UML/SysML tool, EPLAN Software & Services LLC provided access to Electric P8, and JNBridge LLC. provided access to JNBridgePro. Their in-kind support is gratefully acknowledged.

References

1. OMG: Systems Modeling Language v 1.1 (2008), http://www.omg.org/docs/formal/08-11-02.pdf
2. OMG: Meta Object Facility (MOF) Core Specification v 2.0. (2006), http://www.omg.org/docs/formal/06-01-01.pdf
3. OMG: Model Driven Architecture, http://www.omg.org/mda/
4. Kovse, J., Härder, T.: Generic XMI-Based UML Model Transformations. In: Bellahsène, Z., Patel, D., Rolland, C. (eds.) OOIS 2002. LNCS, vol. 2425, pp. 183–190. Springer, Heidelberg (2002)
5. Lubell, J.: From Model to Markup: XML Representation of Product Data. In: XML 2002 Conference (2002)
6. Chen, K., Schaefer, D.: MCAD - ECAD Integration: Overview and Future Research Perspectives. In: ASME International Mechanical Engineering Congress and Exposition, ASME (2007)
7. Alexander, B., Lian, D., Manjula, P.: An Approach to Accessing Product Data Across System and Software Revisions. Advanced Engineering Informatics 22(2), 222–235 (2008)
8. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. Software, IEEE 20(5), 42–45 (2003)
9. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture (2003)
10. Baresi, L., Heckel, R.: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 402–429. Springer, Heidelberg (2002)
11. Giese, H., Levendovszky, T., Vangheluwe, H.: Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 252–262. Springer, Heidelberg (2007)
12. Fujaba: Fujaba Tool Suite, http://wwwcs.uni-paderborn.de/cs/fujaba/
13. MOFLON: MOFLON Homepage, http://moflon.org/
14. Selic, B.: From Model-Driven Development to Model-Driven Engineering. In: Keynote talk at ECRTS 2007 (2007)
15. Vanderperren, Y., Dehaene, W.: SysML and Systems Engineering Applied to UML-Based SoC Design. In: UML-SoC Workshop at 42nd DAC, 2005 (2005)
16. Espinoza, H., Cancila, D., Selic, B., Gérard, S.: Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 98–113. Springer, Heidelberg (2009)
17. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann, San Francisco (2008)

18. EPLAN: EPLAN Electric P8, http://www.eplanusa.com/
19. Modelica: Modelica Language Specification v 3.1. (2009), http://www.modelica.org/documents/ModelicaSpec31.pdf
20. Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., Rieke, J.: Management of Cross-Domain Model Consistency during the Development of Advanced Mechatronic Systems. In: Bergendahl, N., Grimheden, M., Leifer, L., Skogstad, P., Lindemann, U. (eds.) Proceedings of the 17th International Conference on Engineering Design (ICED 2009). Design Methods and Tools, vol. 6, pp. 1–12 (2009)
21. Bajaj, M., Paredis, C.J.J., Rathnam, T., Peak, R.: Federated Product Models for Enabling Simulation-Based Product Lifecycle Management. In: Proceedings of ASME International Mechanical Engineering Congress and Exposition (2005) IMECE2005-81663
22. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electronic Notes in Theoretical Computer Science 148(1), 113–150 (2006)
23. Czarnecki, K.: Overview of Generative Software Development. In: Banâtre, J.-P., Fradet, P., Giavitto, J.-L., Michel, O. (eds.) UPP 2004. LNCS, vol. 3566, pp. 326–341. Springer, Heidelberg (2005)
24. Pop, A., Fritzson, P.: MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language. In: Lightfoot, D.E., Szyperski, C.A. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 211–229. Springer, Heidelberg (2006)
25. Brucker, A.D., Doser, J.: Metamodel-based UML Notations for Domain-specific Languages. In: 4th International Workshop on Software Language Engineering (ATEM 2007) (2007)
26. Weisemöller, I., Schürr, A.: A Comparison of Standard Compliant Ways to Define Domain Specific Languages. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 47–58. Springer, Heidelberg (2008)
27. Kerzhner, A.A., Paredis, C.J.J.: Using Domain Specific Languages to Capture Design Synthesis Knowledge for Model-Based Systems Engineering. In: ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference 2009, ASME (2009) DETC2009–87286
28. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal 45(3), 621–645 (2006)
29. OMG: Meta Object Facility (MOF) 2.0 Query / View / Transformation v1.0. (2008), http://www.omg.org/docs/formal/08-04-03.pdf
30. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
31. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 157–167. Springer, Heidelberg (2000)
32. Königs, A., Schürr, A.: MDI: A rule-based multi-document and tool integration approach. Software and Systems Modeling 5(4), 349–368 (2006)
33. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1), 21–43 (2009)
34. Jobe, J.M., Paredis, C.J.J., Johnson, T.A.: Multi-Aspect Component Models: A Framework for Model Reuse in SysML. In: 2008 ASME International Design Engineering Technical Conferences and Computers in Information Engineering Conference (2008)

35. OpenModelica: The OpenModelica Project, http://www.ida.liu.se/~pelab/modelica/OpenModelica.html
36. Dymola: Dymola, http://www.3ds.com/products/catia/portfolio/dymola
37. NoMagic: MagicDraw, http://www.magicdraw.com
38. JNBridge: JNBridgePro, http://www.jnbridge.com/jnbpro.htm
39. Helms, B., Shea, K., Hoisl, F.: A Framework for Computational Design Synthesis Based on Graph-Grammars and Function-Behavior-Structure. In: 2009 ASME International Design Engineering Technical Conferences and Computers in Information Engineering Conference, ASME (2009) DETC2009–86851

Requirements Engineering in Complex Domains

Matthias Jarke[1], Ralf Klamma[1], Klaus Pohl[2], and Ernst Sikora[2]

[1] RWTH Aachen University & Fraunhofer FIT
{jarke,klamma}@informatik.rwth-aachen.de
[2] University of Duisburg-Essen
{klaus.pohl,ernst.sikora}@paluno.uni-due.de

Abstract. Complexity in the application domains of software-intensive systems is continuously growing due to at least two reasons. Firstly, technical complexity grows as hardware and software have to interact in individual or even communicating embedded systems. Secondly, social complexity grows as the process organizations of the 1990's are gradually being replaced by loosely coupled networks of actors, often organized around community platforms. In this chapter, we discuss recent solution attempts for these two issues individually, and end with speculating about their possible future interaction.

Keywords: abstraction layers, goals, scenarios, architecture, social networks, reflective architectures, web communities.

1 Introduction

As an essential bridge between user and developer concepts, requirements engineering (RE) has increasingly captured the attention of researchers, practitioners, and sometimes even executives and politicians for the past thirty years. Already in the early 1990's, RE was defined as the process of "establishing a vision in context", i.e. the critical role of the context in which a system is developed, operated, and evolved, has been recognized [1]. In two European Basic Research Projects as well as in the German SFB IMPROVE [2], the relations between requirements and the context of the system have been elaborated focusing on aspects such as organization of the context knowledge in different "worlds" of engineering or the interplay of goals and contextual scenarios.

In the 21st century, the basic idea of establishing visions in context remains valid, but the context to be considered and the stakeholders have changed significantly. Firstly, "greenfield" development of completely new systems hardly exists any more. In contrast, changing and new requirements must typically be embedded in large-scale corporate and technical system architectures. Secondly, the pressures on cost and the globalization necessitate internationally distributed "development worlds" with increasing interactions and much stronger needs for precise communication. Third, much of the innovations are no longer coming from the well-structured process organizations (like in the 1990's), but from the edges of shifting cross-organizational networks and flexible service-orientation of which the Web 2.0 movement is probably the best-known example.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 602–620, 2010.

In this paper, we illustrate these new trends and their consequences for requirements engineering researched in two recent projects, one focusing on the co-development of requirements and architectures in complex software-intensive systems (Section 2), the other focusing on requirements engineering in internet communities where the boundaries between users and developers tend to blur as quickly as the membership of the communities themselves (Section 3). Section 4 concludes with some speculation about future confluence of these challenges.

2 Requirements Engineering for Complex Software-Intensive Systems

As Brooks stated in 1987: „The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later." [3]

Today, in many domains such as automotive, avionics, automation, energy, or medicine, innovative product features are realized increasingly by means of software, more precisely through networks of software-intensive, embedded systems. The number of (software-based) system functions and the number of dependencies between these system functions increase rapidly. Hence the complexity of these systems and their software increases as well. The need to develop software-based, innovative functions makes requirements engineering for software-intensive systems more challenging than ever: The requirements engineering process must satisfy strict quality demands which are often enforced even by laws and standards. At the same time, the schedules for developing new systems or, respectively, new system versions are tight, and the development costs must constantly be reduced. Moreover, the proper integration of requirements engineering into the overall development process, especially the intertwining of requirements engineering and architectural design, is becoming an essential factor for successful system development.

In the light of these challenges we have developed the COSMOD-RE method (sCenario and gOal based System development MethOD; see e.g. [4]). COSMOD-RE supports the co-development of requirements and architectural artifacts for software-intensive, embedded systems at multiple layers of abstraction. In the following, we sketch out the cornerstones of COSMOD-RE. In Section 2.1, we provide a brief overview of the key building-blocks of COSMOD-RE. In Section 2.2, we introduce the four abstraction layers which define the backbone of COSMOD-RE. In Section 2.3, we briefly describe the co-development of requirements and architectural artifacts which we support by means of goals and scenarios. Moreover, we illustrate the use of COSMOD-RE using a simplified ACC (adaptive cruise-control system). Section 2.4 provides a brief outlook on the further development of COSMOD-RE.

2.1 COSMOD-RE: Brief Overview

Fig. 1 depicts an overview of the three main building blocks of COSMOD-RE that are described in the next sections: the use of a hierarchy of four abstraction layers, the

support of the intertwined and partly concurrent development of requirements and architectural artifacts by means of co-design-processes, and the use of goals and scenarios to support the refinement of requirements across the abstraction layers as well as the co-development of requirements and architectural artifacts.

Fig. 1. Main building blocks of the COSMOD-RE method

2.2 COSMOD-RE Abstraction Layers

The use of abstraction layers is a well-proven means for problem-solving and, in particular, for dealing with and managing the complexity of software-intensive systems (see e.g. [5]). As Weber and Weisbrod state: "Engineers cannot develop a complex system – such as an up-to-date telematics unit – solely by talking about and dealing with the system requirements that make up the low-level, detailed component specification. On the contrary, developing complex systems from the top down in several layers of granularity is inevitable. When it comes to RE, this observation still holds. Unfortunately, today systematic processing and documentation of higher level requirements and design decisions are insufficient." [6]

In order to support and guide the specification of requirements at different levels of granularity, COSMOD-RE is based on a generalized (or essential) hierarchy of abstraction layers that can be applied to a wide range of domains and systems (see Fig. 2).

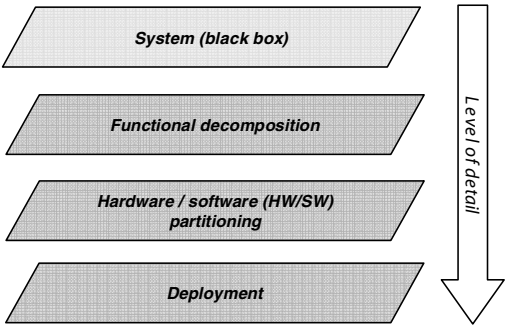


Fig. 2. Hierarchy of four essential abstraction layers defining the backbone of COSMOD-RE

The abstraction layers used in COSMOD-RE are:

- *System layer*: The system layer (highest abstraction layer) defines requirements and architectural artifacts fairly independently of the implementation technology and internal structure (decomposition) of the system. The system is regarded as a black box. This layer thus focuses on the embedding of the system in its environment. The requirements at the system layer are defined mainly from a system usage perspective. At each successive layer, the requirements are refined and additional design concerns are taken into account.
- *Functional decomposition layer*: At the functional decomposition layer, the system is decomposed into coarse-grained, logical building blocks. The requirements for each building block as well as the interrelations and interactions between the building blocks are defined at this layer. The functional decomposition layer aims at supporting problem understanding and problem decomposition.
- *Hardware/software (HW/SW) partitioning layer*: At the hardware/software partitioning layer, the system is decomposed into coarse-grained, technical building blocks, i.e. hardware and software building blocks. The requirements for each building block as well as the interrelations and interactions between the hardware and software building blocks are defined. At this layer, the decision is made which system properties are realised through hardware components and which ones are realised through software components.
- *Deployment layer*: At the deployment layer, the deployment of the system (i.e. the application software and hardware) into a hardware and software platform is defined. Such a platform typically consists of a set of physical units which are connected by physical networks and are equipped with basic system software. Hence, at this layer, the decision is made which application software or hardware component is deployed to which physical unit. This entails refining the requirements with regard to the specifics of each physical unit (such as processing speed and memory, input and output facilities, operating system, communication software, middleware etc. of the respective physical unit).

This hierarchy can be refined or simplified, if needed. The specific hierarchy of abstraction layers used in a project should take into account characteristic properties

of the domain, the organization, or even the project itself (such as system complexity, desired reusability etc.). Using such a hierarchy of abstraction layers offers the following advantages for the development of complex, software-intensive systems (see [7] for more details):

- *Hierarchical (problem) decomposition*: When using abstraction layers, the problem of defining detailed requirements for a software-intensive system is decomposed into a set of smaller problems of defining detailed requirements for individual components and their interactions. Clearly, specifying, for instance, the behavioral requirements for a component (e.g. the requirements for the brake control software) is a simpler task than specifying the behavioral requirements for the entire vehicle at the same level of detail simultaneously.
- *Requirements stability*: Requirements and, partly, architectural solutions at higher abstraction layers are defined independently of their technical realizations e.g. by hardware and software components. Since requirements defined at the system layer are fairly independent of the technical solution, they are typically not affected by changes in the technical realization, such as changes at the hardware/software or the deployment layer. The effects of the abstraction layers are similar to the positive effects of the differentiation between the "essence of a system" and the "incarnation of a system" (technology-independent and technology-dependent requirements) in Essential Systems Analysis [8].
- *Traceability and rationale*: Requirements defined at lower abstraction layers can be related e.g. by means of "refines" relationships to requirements defined at higher abstraction layers. The requirements defined at the lower abstraction layers can be traced back to higher-level requirements. The requirements defined at higher abstraction layers thus provide rationale for the requirements defined at the lower abstraction layers. Among other things, this improves the traceability of the requirements and contributes to their comprehensibility.

When developing requirements at multiple abstraction layers, it becomes obvious that requirements engineering and architectural design are inevitably intertwined (see e.g. [9] [10]) as discussed in the next section.

2.3 Co-development of Requirements and Architectural Artifacts

When developing a complex system, the stakeholders have to accomplish two different but closely related tasks (see e.g. [11]):

- *Refinement of the requirements*: The stakeholders have to refine high-level requirements into detailed requirements which contain enough details to facilitate the implementation and quality assurance (e.g. testing) of the system.
- *Decomposition of the system*: The stakeholders have to decompose the overall system into a set of interacting parts (sub-systems or components), i.e. the stakeholders have to define a detailed architecture which satisfies the defined (detailed) requirements.

Clearly, the requirements influence and partly even determine the architecture. However, design decisions (e.g. the choice of a specific architectural solution for the system) strongly influence the refinement of high-level requirements into detailed

(implementable and testable) requirements. In other words, the refinement of the requirements depends (partly) on the design choices taken. In addition, an innovative architectural solution can even lead to entirely new (high-level) requirements.

As suggested, for example, by Nagl and his co-authors (see e.g. [12]), development methods and tools should support a tight integration between requirements engineering and architectural design. Still, existing development methods do not foster the intertwined development of requirements and architecture. Refining requirements without a systematic exploration of possible design options bears the danger that important design decisions are made implicitly. In other words, stakeholders often hide design decisions taken (explicitly or implicitly) in the detailed requirements. Such implicit design decisions often rule out other (and perhaps better) solutions. In order to avoid such problems (implicit design decisions), the COSMOD-RE method fosters and systematically supports the intertwined development of requirements and architectural artifacts.

2.3.1 A Co-design Process

COSMOD-RE supports the intertwined and partly concurrent development of requirements and architectural artifacts by means of so-called co-design processes. Fig. 3 depicts the structure of the co-design processes defined by COSMOD-RE. A co-design process in COSMOD-RE supports the development of requirements artifacts at an abstraction layer L_i and the alignment of these requirements artifacts with the architectural solution at the abstraction layer L_{i+1}.

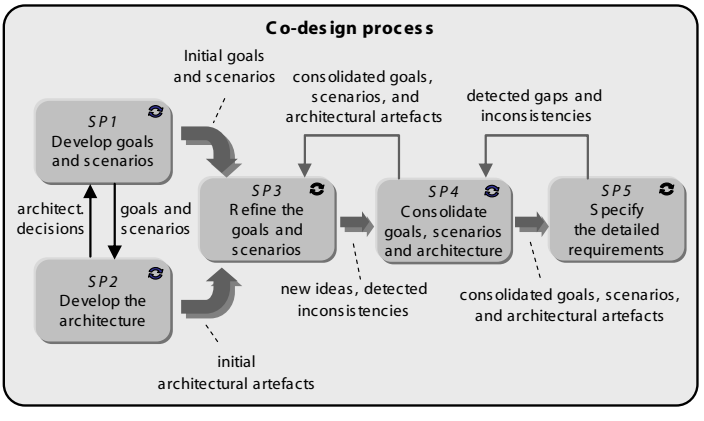


Fig. 3. Overview of the sub-processes of a co-design process

The goals of the five sub-processes depicted in Fig. 3 can be characterized as follows:

- Sub-process SP_1 supports the development of initial goals and scenarios at the layer L_i (see Section 2.3.2).
- Sub-process SP_2 supports the development of initial architectural artifacts at the layer L_{i+1}. Sub-processes SP_1 and SP_2 can be performed in an intertwined manner or, partly, in parallel.

- Sub-process SP_3 supports refinement of the goals and scenarios defined at layer L_i based on the architecture defined at layer L_{i+1}. The refinement results in goals and scenarios at layer L_{i+1}. In addition, the responsibilities for satisfying the goals and scenarios are assigned to the elements of the architecture. Thereby, the detection of mismatches between the requirements defined at layer L_i and the architecture at layer L_{i+1} is supported (such as goals that the architecture cannot satisfy or architectural elements not justified by system goals).
- Sub-process SP_4 is responsible for reconciling the requirements and the architecture at the two neighbouring layers L_i and L_{i+1}. It results in corrections and changes applied to the requirements artefacts and the architectural artefacts at both layers.
- In sub-process SP_5, the detailed (solution-oriented) requirements at layer L_i are specified based on the results obtained from the other sub-processes, i.e. consolidated goals, scenarios, and (coarse-grained) architectural artifacts.

In the following, we motivate the use of goals and scenarios for supporting the co-development of requirements and architectural artifacts as well as the refinement of requirements across multiple abstraction layers. The use of goals and scenarios for these purposes is illustrated by a brief example (Section 2.3.3).

2.3.2 Goals and Scenarios

COSMOD-RE uses goals and scenarios to support the co-development of requirements and architectural artifacts as well as the refinement of system requirements into component requirements. Goals document stakeholder intentions and thereby refine the system vision into verifiable system objectives. Goals are typically solution-neutral. For example, they do not prescribe or assume the use of a specific technology. In addition to goals, scenarios are used to document concrete examples of system usage which lead to the fulfilment or unfulfilment of a defined goal. Typically, a scenario is defined as a sequence of interaction steps. Goals and scenarios prevent the stakeholders from making premature design decisions or focusing on solution details too early. Moreover, goals and scenarios define a sound basis for identifying and exploring possible solutions in the downstream development activities:

- *Definition of an initial, coarse-grained architecture*: Typically, as soon as the system goals and scenarios are defined and agreed, an initial, coarse system architecture can be developed. Therefore, goals and scenarios are well suited to support the co-development of requirements and architectural artifacts.
- *Definition of solution-oriented system requirements*: The defined, agreed, and verified system goals and scenarios (along with the initial, coarse architecture) provide a sound basis for defining detailed, solution-oriented requirements for the system. The resulting, detailed system requirements are typically more stable if agreed and consolidated goals and scenarios are used as input.
- *Definition of component requirements*: Typically, it is significantly easier to refine goals and scenarios than refining, for instance, a data model, a functional model, or a behavioral model. Hence, in COSMOD-RE, system goals and system scenarios are refined into component goals and components scenarios prior to defining (solution-oriented) component requirements. In this way, component goals and component scenarios are provided to support the definition of (solution-oriented) component requirements.

2.3.3 An Example

We illustrate the development and refinement of goals and scenarios across two abstraction layers (the system layer and the functional decomposition layer; see Section 2.2) using a simplified example of an adaptive cruise control (ACC) system.

We document goals and their relationships using the KAOS goal model [13]. A simplified goal model for the ACC system is depicted in Fig. 4 on the left. In addition, details about the defined goals are documented using a goal template (see [7] for details). Scenarios are documented using a use case diagram, use case templates, and sequence diagrams. The use cases identified for the ACC system to fulfill the identified system goals are depicted in Fig. 4 (on the right-hand side).

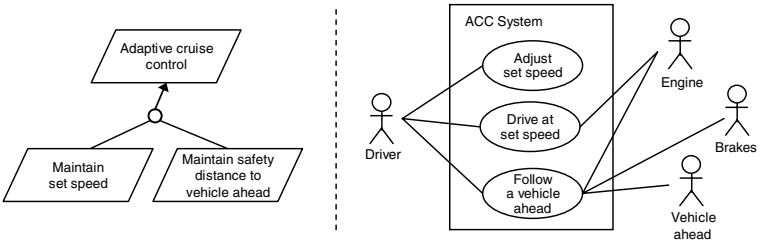


Fig. 4. Examples of a goal model (left) and a use case diagram (right) for the ACC system

Based on the system goals and scenarios, the system architects define an initial, coarse architecture for the ACC system that satisfies the defined goals and scenarios. The initial architecture depicted in Fig. 5 defines the major functional components of the ACC system and their interfaces.

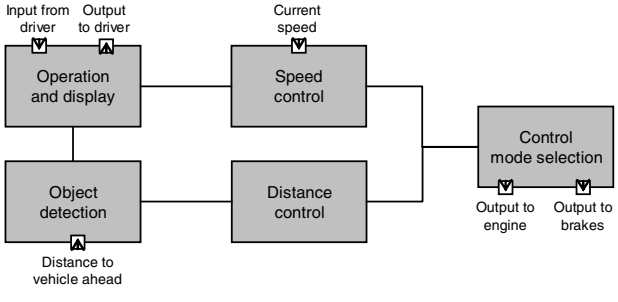


Fig. 5. Example of an initial, coarse architecture for the ACC system

Based on the initial architecture, the system goals and scenarios are (if required) refined and related to the architectural elements. Thereby, mismatches between the requirements (goals and scenarios) and the suggested architectural solution can be detected early. For instance, an architectural component may have no associated goals and thus no justification for its existence. Fig. 6 depicts the refinement of a system goal and the assignment of the resulting sub-goals to individual, architectural components.

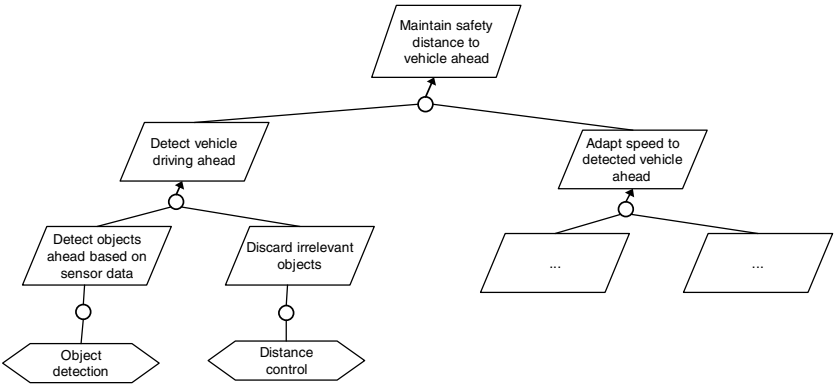


Fig. 6. Example of refining a system goal into a set of component goals

Based on the relations between the refined goals and scenarios and the initial architecture, problems and inconsistencies are detected and removed in order to align the goals, scenarios, and architecture. Furthermore, the alignment often stimulates new ideas for goals, scenarios, or architectural solutions (see [7] for more details on aligning goals, scenarios, and the architecture). The aligned goals, scenarios, and architectural artifacts are subsequently used as a basis for defining the detailed, solution-oriented system requirements (see [7] for details).

2.4 Further Reading and Outlook

Additional details on COSMOD-RE can be found in several publications. The idea of co-developing requirements and architecture is explained in more detail in [14]. A comprehensive description of COSMOD-RE has been included in a recent textbook on requirements engineering, see [7].

During the construction of COSMOD-RE, an initial industrial evaluation of the method has been performed. During this evaluation, COSMOD-RE was successfully applied to various industrial examples. Presently, a systematic evaluation of in a number of different domains such as automotive, avionics, or medical technology is performed. Initial results of the evaluation of COSMOD-RE in the automotive domain are reported in [15].

To support automated, formal reasoning in COSMOD-RE, parts of the method are being formalized [16]. Furthermore, COSMOD-RE is being integrated into a seamless, model-based, overall development process for software-intensive embedded systems. This work is conducted within the German Innovation Alliance SPES 2020 (Software Platform Embedded Systems).

3 Requirements Engineering in Community Networks

Business process modeling and business process management, including their software equivalents of ERP and workflow systems, have been the dominant

paradigm of the 1990's. Since the turn of the century, however, the growing role of the Internet begins to lead to a shift in emphasis, which may lead beyond well-structured process organizations to much more flexible capability-based network organizations. They blur organizational boundaries and emphasize *generic platforms*, *innovation at the margin* of the network, and *rapid reorganization* over central process planning. Social software has begun to play a serious role in many enterprises, as a medium of knowledge management, project management, or public relations. Important contributors to this success are *low entry barriers*, *easy usability*, and *great flexibility*. Thus, community networks have become an informal, not necessarily friendly counterpart to the structured systems described in section 2.

In stark contrast to the easy initial entry, the long-term structuring of these information networks, e.g. in order to interface them with organizational requirements management, is far more difficult. Bottom-up voices from "community of practice" in Web 2.0 media differ strongly from contract-based requirements engineering documents with their many formal and semi-formal interdependencies. An architecture intended to support community information systems on this basis requires particular flexibility. A community needs to be able to *observe* itself, to *analyze* and maybe even *simulate* its behavior, in order to evolve its rules of cooperation. In short, it needs *reflective capabilities*. Only then it can survive as a community, and even interact with more formal parts of the structured process organizations. In this section, we shall be interested in the question how these properties can be achieved, such that – in the long run – the present disparity between the structured product family and process world of formal organizations can reach a more fruitful interplay with the social network and software environment.

Our approach originates from an operational theory of media called *Transcriptivity Theory* [17]. It describes the operational semantics of media artifacts with three basic operations *transcription* (of sets of media into new media), *localization* (filtering and adapting received media to your own culture) and *addressing* (media to specific target groups of readers). In our abstract software architecture *ATLAS* (*Architecture for Transcription, Localization, and Addressing Systems*) [18], transcriptivity theory has been re-interpreted as a design theory for information systems in which scalable and interoperable repositories on top of databases support communities by web service technologies for multimedia content and metadata management.

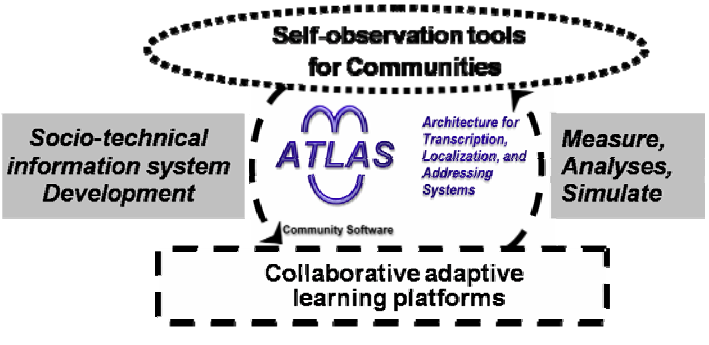


Fig. 7. The ATLAS Development Model

In its reflective conception, ATLAS-based community information systems are tightly interwoven with a set of media-centric self-monitoring tools. Hence, communities can constantly *measure*, *analyze* and *simulate* their activities to improve the understanding of their needs, in turn simplifying the collaboration between developers and users (cf. Fig. 7). In the future, all community design and engineering activities should be carried out by the community members, regardless of their technical knowledge. In the sequel, we illustrate challenges and solutions for this concept using a specific example of network organization, cooperative learning.

The ROLE (Responsive Open Learning Environments) architecture (cf. Fig. 8) is an example application of the ATLAS architecture specialized on the development of Internet based personal learning environments. ROLE is developed in the context of an EU-funded Integrated Project with the same title.

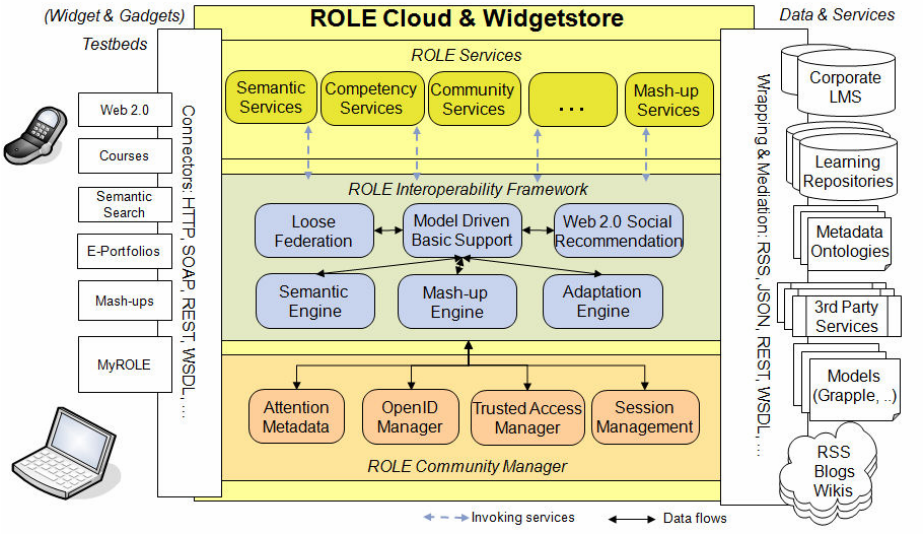


Fig. 8. The ROLE Architecture

The design of the *ROLE Cloud* – the open network of learning resources and services -- is based on a lightweight service-oriented architecture fostering communication and data provision. The *ROLE Community Manager* provides a basic service layer for privacy and trust mechanisms as well as traceable learning histories for learner communities. The *ROLE Interoperability Framework* improves the support for distributed third party developers from industrial as well as open source contexts to foster collective enhancements of the ROLE framework and to open new opportunities for SMEs in the cooperative learning domain.

The ROLE methodologies and specifications evaluate solutions for deployment, repurposing, customization, personalization, and validation in the *ROLE test-bed frameworks*, before outreaching to further communities. We aim at a self-sustaining

networked community of online learners and developers, at developing a best practice sharing strategy for virtual, collaborative learning activities, and at examining characteristics and circumstances of such a learning network. A special focus is thereby put on the elicitation of early phase requirements in communities. In the next subsections, we first discuss the special RE needs of such communities which are quite different from those of formal organizations. We then structure these needs using Eric Yu's i* model, and finally discuss a couple of community monitoring and analysis tools we have developed to support the reflection process.

3.1 Community of Practice

The conceptualization of RE for communities of practice (CoP) requires a thorough investigation of the existing forces, which can influence community member behavior. Communities are influenced by many external factors called *disturbances*. These disturbances may have a negative, positive or neutral influence on the learning processes within the community [19]. Without disturbances, communities are in danger of falling into social or cognitive lock-in situations [20, 21]. Usually, the disturbances are coming from outside but are changing the community inside. Thus, the evolution of communities takes place in a lifelong loop.

A CoP is characterized by three dimensions introduced by Wenger [22]:

Mutual engagement (ME): Community members are required to be engaged in interaction within their community. Hereby, membership in a community is not just belonging to one organization, and a comunity is not only a set of members having personal contacts with other members.

Joint enterprises (JE): The common goal of a community of practice, which binds members together is the result of a collective process of negotiation which reflects the full complexity of mutual engagement.

Shared repertoire (SR): The communal resources include routines, words, stories, gestures, symbols, genres, actions, tools, ways of doing things, concepts, etc. that the community has produced or adopted during its existence, as part of its practice.

A community is thus an open group of people who share a concern or a passion and who interact regularly about it [22]. Two aspects are combined: the social practice of the community as a collective phenomenon and the identity of its members as an individual phenomenon. Individual learning is inherent in the processes of social participation in the community. Knowledge and learning are not abstract models but relations *"between a person and the world"* [23] or *"among people engaged in an activity"* [24]. Individual learning is mainly based on *"legitimate peripheral participation"* [22]. During the participation process, an individual might enter the community as a beginner at the periphery and then gain a more central position over time by cognitive apprenticeship. This acquisition process leads to an intensified inclusion in the social practice of the community. Learning is based on this process of inclusion of outsiders. The communities of practice themselves can be seen as *"shared histories of learning"* [22]. The mechanism of (social) identification of individuals in the social context of the community plays a key role for community

formation as well as community survival but there does not yet exist a stable lifecycle theory how to achieve continued success of a community; our hope is that a more systematic community RE process can contribute to the development of such a lifecycle support.

3.2 Towards a Community RE Process

To map learning communities participating in RE to community concepts, we need a theory that explains socio-psychological aspects, which influence community members through relations between human agents, technologies and resources. For this purpose, Actor-network-theory (ANT) [25], which makes no distinction between human and non-human actors, can be adopted, because it intertwines actions, influences, and results of actions independently if automated or human.

In addition to this action-related approach to network modeling, RE also needs to capture intentions of (usually human) actors and the strategic dependencies which are considered as the long-term basis for cooperation in a community of specialists. To capture this intentional aspect of networks, we chose the i* framework [26], which enables the description of strategic relations between actors within a particular socio-technical system in a clear way [27]. In i* we find the following premises: Agents are dependent on other agents. They act intentionally, because they follow goals, have beliefs, competences, commitments, needs and desires. At the same time, agents are strategic actors, because they have to cooperate to reach their goals. Intentional dependencies can be made explicit with i* to disclose the reasons behind observable processes. In the following we provide a brief overview of i* modeling components.

In i*, strategic dependencies (SD) are used for modeling intentional, strategic relationships among actors in form of an actor diagram. Strategic rationale (SR) models cover the individual rationale behind dependencies, and analyzes alternatives and dependencies fulfillment by a goal diagram. The model of our community RE process is an example of strategic rationale. The main syntax elements of i* are: *Actors*, *Actor Associations*, *Goals*, *Softgoals*, *Tasks*, *Resources* and *Links*. An *actor* can be any active entity that carries out actions to achieve goals by exercising its know-how. An *agent* is an actor with concrete, physical manifestations, such as a human individual or a software system. Actors in SR contain intentional *boundaries*. All elements within such a boundary are explicitly desired by an actor. The fulfillment of goals within boundaries can depend on intentions of other actors and is represented as *strategic dependencies*. Such a dependency consists of a *dependee*, who is required to fulfill a *dependum* by a *depender*. Links in SR can be of three types: *decomposition*, *means-ends* and *contribution*. A decomposition link describes a decomposition of a task in subtasks, subgoals, resources or softgoals. Means-ends-links bind a goal to achieve with a task, which has to be executed for attaining the goal. Finally, contribution links connect softgoals to tasks.

An i* model for RE within a community of practice (*CoP*) is presented in Fig. 9. The main idea is firstly to combine analysis of community-generated content and system usage, and secondly to provide the *Social System* of CoP members with a service for expressing their requirements.

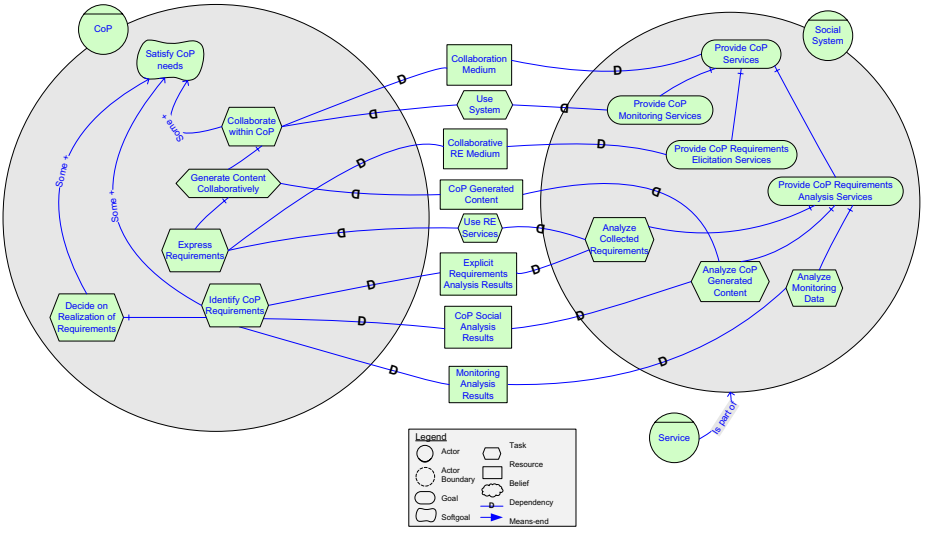


Fig. 9. Model of Requirements Engineering in CoPs

CoP and *Social System* are modeled as agents (big dark circles) according to i* terminology. The *Social System* is a software environment used by CoP members in order to fulfill the softgoal of satisfying CoP needs (Joint Enterprise). Thus, the *Social System* presents a composition of different Services, depending on the community and its practices. The *Social System* should additionally provide services specified for the RE process: *Provide CoP Monitoring Services*, *Provide CoP Requirements Analysis Services*, *Provide CoP and Requirements Elicitation Services*.

Any community requires a shared repertoire, which can be created and enriched, as community members collaborate with each other (*generate content collaboratively*). Therefore, the *Social System* should offer CoP services as a collaboration medium. However, the availability of a service is not enough for collaboration. According to the definition of the mutual engagement dimension of CoP, community members are required to be engaged in interaction; thus, they do not only consume, but also *generate content collaboratively*. Hereby, CoP-generated content can be applied to get insights in social characteristics of the community and thus *identify CoP requirements*. Log data of system usage can serve as the next source for requirements mining. For this reason, the *Social System* provides a *CoP monitoring service*, which collects data for later analysis.

In our ATLAS prototype, the services of the *Social System* are technically delivered though extensions to LAS, a Lightweight Application Server for prototyping, testing and hosting new community web services [29]. In the remainder of this section, we briefly discuss two of these interactive services intended to assist the RE process in a community, one for the structured monitoring interactive assessment of the quality of existing services under special consideration of mobile services (MobSOS [28]), the other for the self-analysis of undesirable developments in the community social network structure itself (PALADIN [19]). Both services have already been successfully used in numerous research and practice projects.

The main purpose of MobSOS is the integrated support for (mobile) web service quality measurement and evaluation. The underlying success model was inspired by the widely used DeLone/McLean IS success model [30]. MobSOS offers two modules for the acquisition of model test data: monitoring and user surveys. The monitoring module logs communication between users and services together with context information, thus enabling context-dependent quality analysis, according to the data model shown in Fig. 10. All other quality data required by the success model are collected by the user survey module.

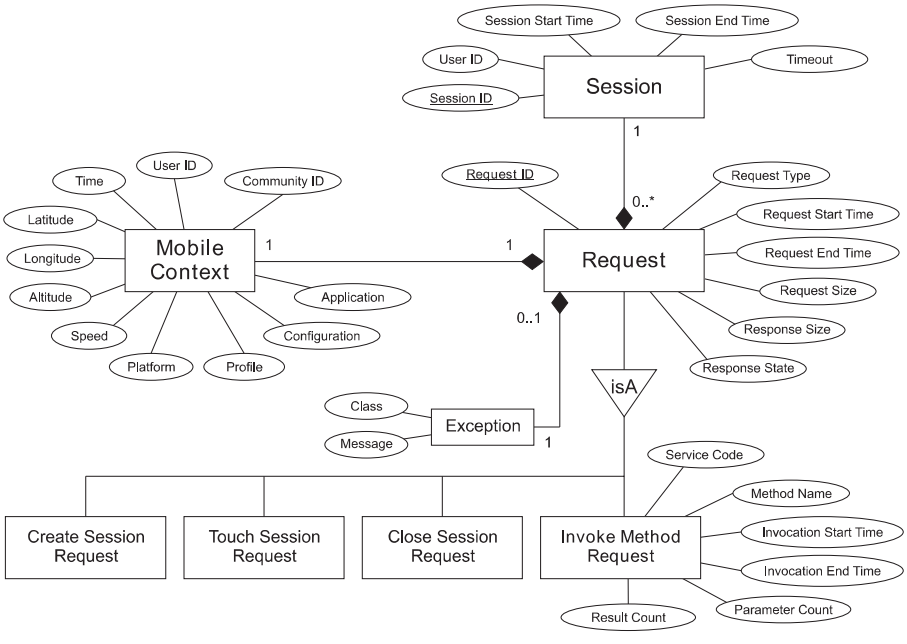


Fig. 10. Monitored Community System Usage and Context in MobSOS

The PALADIN service has been developed for the detection of multidimensional disturbance patterns in communities. It includes a pattern structure definition, the *Formal Expression Language for Patterns* (*FELP*), and algorithms for the application of the patterns on the data. Each pattern consists of a *name*, a *disturbance*, a *description*, *forces*, *force relations*, a *solution*, a *rationale*, and *pattern relations*. The *forces* are relevant actors of the pattern; the *forces relations* are relations between actors. The *solution* provides the advices to solve the pattern situation. The relations are used for reasoning about the forces and the disturbances. PALADIN has been successfully applied in the detection of community patterns in ten thousands of mailing lists. As an example (cf. Fig. 11), the *conversationalist pattern* identifies members (dark circles) in communities which are engaging also in conversations by participating in threads without having started them (comparable to “flame warriors”). The conversationalists discuss ideas, carry on conversations and share opinions. The

conversationalist pattern has two relations - `rcnv1 = (own thread, conversationalist, thread)` which relates the member to the threads which he/she has started and `rcnv2 = (post thread, conversationalist, thread)` which relates the member to the threads where he/she has posted. The conditions for a member to be a conversationalist are:

```
The member has started at least one thread; AND the
member has posted in at least one thread started by
other members; AND the maximum number of messages,
part of a conversationalist's thread, must be
greater than n (in the figure: n = 4); AND the
member has posted more than k messages overall.
```

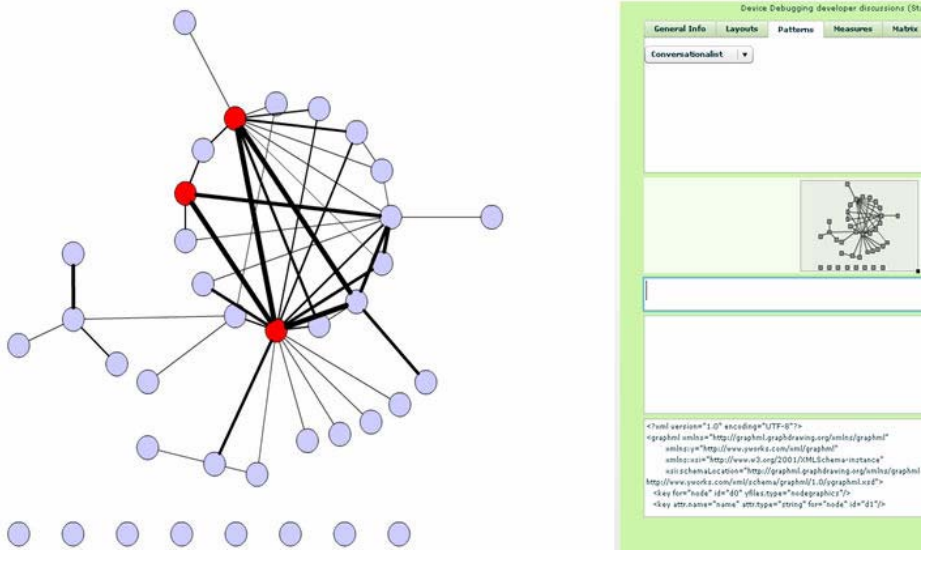


Fig. 11. Matching the Conversationalist Community Pattern in PALADIN

Beside these two implicit methods of requirements identification, the ATLAS environment also provides communities with a mobile collaborative RE service, where community members can express requirements explicitly in a multi-media assisted scenario-based approach.

From these varied sources and RE services, the generated pool of requirements can be huge and diverse. Hence, the *Social System* (cf. Fig. 9) has to analyze collected requirements first and then reports the results to the *CoP*. Using the results from social, system usage and requirements analysis, it is the task of the *CoP* to prioritize requirements and to decide on their particular realization.

4 Discussion and Outlook

The two approaches described in the previous sections demonstrate that, with the ever-growing spread of software and information technologies in all branches of engineering, business organizations and society, requirements engineering approaches have diversified.

The COSMOD-RE method presented in Section 2 addresses specific issues in the engineering of complex, technical systems. The main goal of the COSMOD-RE method is to support the tightly intertwined development of the requirements and the architecture (structure) of complex technical systems from coarse-grained system requirements down to detailed hardware and software component requirements. The approach has the advantage that the development of requirements and architectural models is structured by means of well-defined abstraction layers and the engineers are supported in choosing the appropriate level of abstraction for their models throughout the process.

The community RE approach presented in Section 3 focuses on involving a diversified community in the system development process. In contrast to COSMOD-RE, the main emphasis of the community RE approach is in fact the community itself. The community RE approach has the advantage of a very direct impact of new user and developer ideas on not just the evolution of the system but also of its user community. Fresh ideas from the margin of the community network can assist in radical innovation, and there is some evidence from practice applications that the additional community self-control offered by the monitoring services can reduce to some degree the brittleness many community systems have suffered from in the past.

The two approaches described in this chapter address two different aspects of complexity in requirements engineering: the complexity of the user and developer community and the complexity of the system itself. While in this contribution, the complexity of the community and the complexity of the system are treated separately, in the near future, the need arises to address both kinds of complexity in coherent requirements engineering approaches: With the upcoming, converged future internet, embedded systems controlling physical processes will be tightly integrated with information systems providing innovative services to their users based on the up-to-date information about the physical world. For instance, in the energy domain, a network of smart metering devices can provide real-time information about power consumption. Business information systems can exploit this data to offer advanced analysis services for decision makers in energy supply and trading companies. Similarly, in logistics, sensor networks can capture information about the location and status of physical goods in order to support advanced services for logistics planning.

At the same time rigid organization forms will dissolve into often loosely coupled and rapidly changing community networks. Organizations who have their roots in technical domains (such telecommunications or power supply infrastructure) are being transformed into service providers offering information-intensive software services to a complex user community via the internet. One of greatest challenges for the next decade will be to support this change by providing development approaches that incorporate the ideas, needs, and wishes of user communities as well as the technical knowledge of domain experts in order to successfully develop the next generation of software-intensive systems in the converged, future internet.

References

1. Jarke, M., Pohl, K.: Establishing Visions in Context – Towards a Model of Requirements Processes. In: Proc. 14th Intl. Conference on Information Systems, pp. 23–34 (1993)
2. Nagl, M., Marquardt, W.: Collaborative and Distributed Chemical Engineering – From Understanding to Substantial Design Process Support – Results of the IMPROVE Project. LNCS, vol. 4970. Springer, Heidelberg (2008)
3. Brooks, F.: No Silver Bullet – Essence and Accidents of Software Engineering. IEEE Computer 20(4), 10–19 (1987)
4. Pohl, K., Sikora, E.: COSMOD-RE: Supporting the Co-Design of Requirements and Architectural Artifacts. In: Proc. 15th IEEE Intl. Requirements Engineering Conference, pp. 258–261. IEEE Computer Society, Los Alamitos (2007)
5. Leveson, N.: Intent Specifications – An Approach to Building Human-Centered Specifications. IEEE Transactions on Software Engineering 26(1), 15–35 (2000)
6. Weber, M., Weisbrod, J.: Requirements Engineering in Automotive Development – Experiences and Challenges. IEEE Software 20(1), 16–24 (2003)
7. Pohl, K.: Requirements Engineering – Fundamentals, Principles, Techniques. Springer, Heidelberg (to appear, 2010)
8. McMenamin, S., Palmer, J.: Essential Systems Analysis. Prentice Hall, London (1984)
9. Swartout, W., Balzer, R.: On the Inevitable Intertwining of Specification and Implementation. Communications of the ACM 25(7), 438–440 (1982)
10. Nuseibeh, B.: Weaving Together Requirements and Architectures. IEEE Computer 34(3), 115–117 (2001)
11. Harel, D., Pnueli, A.: On the Development of Reactive Systems. NATO ASI Series, vol. F13, pp. 477–498. Springer, Heidelberg (1985)
12. Kohring, C., Lefering, M., Nagl, M.: A Requirements Engineering Environment within a Tightly Integrated SDE. Requirements Engineering, vol. 1, pp. 137–156. Springer, Heidelberg (1996)
13. Van Lamsweerde, A.: Requirements Engineering – From System Goals to UML Models to Software Specifications. Wiley, Chichester (2009)
14. Pohl, K., Sikora, E.: Structuring the Co-Design of Requirements and Architecture. In: Proc. 13th Intl. Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2007), Trondheim, Norway (2007)
15. Sikora, E., Pohl, K.: Evaluation eines modellbasierten Requirements-Engineering-Ansatzes für den Einsatz in der Motorsteuerungs-Domäne. In: Proc. Erster Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme (ENVISION 2020). Gesellschaft für Informatik, LNI, vol. 160 (2010)
16. Sikora, E., Daun, M., Pohl, K.: Supporting the Consistent Specification of Scenarios Across Multiple Abstraction Levels. In: 16th Intl. Working Conference on Requirements Engineering – Foundation for Software Quality (REFSQ 2010) (2010)
17. Jäger, L.: Transkriptivität - Zur medialen Logik der kulturellen Semantik. In: Jäger, L., Stanitzek, G. (eds.) Transkribieren - Medien/Lektüre, Fink, München, pp. 19–41 (2002)
18. Jarke, M., Klamma, R.: Reflective community information systems. In: Manolopoulas, Y., et al. (ed.) ICEIS 2006. LNBIP, vol. 3, pp. 17–28. Springer, Heidelberg (2006)
19. Klamma, R., Spaniol, M., Denev, D.: PALADIN: A Pattern Based Approach to Knowledge Discovery in Digital Social Networks. In: Tochtermann, K., Maurer, H. (eds.) Proceedings of I-KNOW 2006, 6th International Conference on Knowledge Management, Graz, Austria. J.UCS (Journal of Universal Computer Science) Proceedings, pp. 457–464. Springer, Heidelberg (2006)

20. Granovetter, M.S.: The strength of weak ties: A network theory revisited. In: Lin, P.M.N. (ed.) Social Structure and Network Analysis, pp. 105–130. Sage, Beverly Hills (1982)
21. Krippendorf, K.: Some principles of information storage and retrieval in society. General Systems 20, 15–35 (1975)
22. Lave, J., Wenger, E.: Situated Learning: Legimate Peripheral Participation. Cambridge University Press, Cambridge (1991)
23. Duguid, P.: The Art of Knowing: Social and Tacit Dimensions of Knowledge and the Limits of the Community of Practice. Information Society 21(2), 109–118 (2005)
24. Østerlund, C., Carlile, P.: How practice matters: A relational view of knowledge sharing. In: Huysman, M., Wenger, E., Wulf, V. (eds.) Communities and Technologies - Proceedings of the First International Conference on Communities and Technologies (C&T 2003), pp. 1–22. Kluwer Academic Publishers, Dordrecht (2003)
25. Latour, B.: On recalling ANT. In: Law, J., Hassard, J. (eds.) Actor-Network Theory and After, Oxford, pp. 15–25 (1999)
26. Yu, E.: Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In: Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE 1997), Washington D.C., USA, January 6-8, pp. 226–235 (1997)
27. Bryl, V., Giorgini, P., Mylopoulos, J.: Designing socio-technical systems: from stakeholder goals to social networks. In: Requirements Engineering, vol. 14(1), pp. 47–70. Springer, New York (2009)
28. Renzel, D., Klamma, R., Spaniol, M.: MobSOS - A Testbed for Mobile Multimedia Community Services. In: 9th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2008), Klagenfurt, Austria (May 2008)
29. Spaniol, M., Klamma, R., Janssen, H., Renzel, D.: LAS: A Lightweight Application Server for MPEG-7 Services in Community Engines. In: Tochtermann, K., Maurer, H. (eds.) Proceedings of I-KNOW 2006, 6th International Conference on Knowledge Management, Graz, Austria. J.UCS Proceedings, pp. 592–599. Springer, Heidelberg (2006)
30. DeLone, W.D., McLean, E.R.: Information Systems Success: The Quest for the Dependent Variable. Information Systems Research 3(1), 60–95 (1992)

Tool Support for Dynamic Development Processes

Thomas Heer[1], Markus Heller[2], Bernhard Westfechtel[3], and René Wörzberger[1]

[1] Department of Computer Science 3, RWTH Aachen University, D-52056 Aachen
{heer,woerzberger}@i3.informatik.rwth-aachen.de
[2] SAP Research, CEC Karlsruhe, Vincenz-Priessnitz-Straße 1, D-76131 Karlsruhe
markus.heller@sap.com
[3] Applied Computer Science I, University of Bayreuth, D-95440 Bayreuth
bernhard.westfechtel@uni-bayreuth.de

Abstract. Development processes in engineering disciplines are highly dynamic. Since development projects cannot be planned completely in advance, the process to be executed changes at run time. We present a process management system which seamlessly integrates planning and enactment. The system manages processes at the project management level, but goes beyond the functionality of project management systems inasmuch as it both monitors and controls development processes and supports enactment of tasks through a work environment. However, the process management system does not provide process automation as performed in workflow management systems. Therefore, we have developed tools for integrating process management and workflow management such that repetitive fragments of the overall development process may be enacted in workflow management systems and monitored in the process management system. Even in the case of repetitive process fragments, the need for deviations from the workflow definition may occur while a workflow is being enacted. Thus, we have also realized a tool which allows to perform dynamic changes of workflows during enactment. Altogether, dynamic development processes are supported through a synergistic combination of process and workflow management systems, integrating process planning and enactment.

Keywords: Project Management, Process Management, Dynamic Changes, Workflow Management.

1 Introduction

Development processes in engineering disciplines such as mechanical, chemical, and software engineering are highly dynamic. Typically, a development project may not be defined and planned completely in advance. Rather, at project runtime the process may have to be changed for a number of different reasons: The steps to be executed may depend on the structure of the product to be developed, problems may be detected which require feedback to earlier steps of the process, the knowledge of the process to be performed is incomplete or imprecise, etc. Thus, *dynamic changes* of the process have to be performed at project runtime, particularly in the case of long-term projects lasting for months or years.

This paper reports on *tool support* for dynamic development processes. It is based on long-term work which has been performed in the group of Manfred Nagl at RWTH

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 621–654, 2010.

Table 1. Comparison of solutions for development process management

	(1) Project Management Systems	(2) Integrated Project and Workflow Management Systems	(3) Process Management Systems	(4) Integrated Process and Workflow Management Systems	(5) Workflow Management Systems
Tools	• MS Project, Primavera, RPLan, et al.	• MILOS, IPPM, et al.	• AHEAD	• AHEAD + Shark	• Shark, Staffware, InConcert, WPS, et al.
Data	• Project plan	• Project plan • Workflow instances	• Dynamic task net	• Dynamic task net • Workflow instances	• Workflow instances
Characteristics	• Only planning • No enactment • No modeling of products	• Planning on project level • Enactment of individual subprocesses	• Dynamic task net represents plan and enactment state	• Workflow instances represent partially automated subprocesses	• Enactment of subprocesses • Modeling of overall development process infeasible
Consequences	• Monitoring and control difficult	• Mapping of status of workflows to project plan difficult	• Planning and controlling well supported • No automation of processes	• Automation of subprocesses, but • Less flexibility in workflow-managed subprocesses	• Subprocesses unconnected • No planning and scheduling support
	Project Planning				Process Enactment

Aachen University. Among these projects, the Collaborative Research Center *IMPROVE* [1], which was dedicated to models and tools for development processes in chemical engineering, played a dominant role. In addition, other engineering disciplines such as mechanical engineering and software engineering were studied, as well. In all of these research efforts, the dynamics of development processes have been a recurring theme which was addressed by a variety of approaches.

In the current paper, we present an overview of these approaches, which complement each other in order to provide comprehensive tool support for dynamic development processes. We structure the presentation with the help of Table 1. In the sequel, we first discuss related work and then describe the contributions of our own work.

1.1 Related Work

Project Management Systems. A development process is always enacted in the form of a development project which has a restricted set of human resources and given external deadlines. It is common practice to use *project management systems* for project planning and scheduling (column (1) of Table 1). Prominent examples for proprietary project management tools are Microsoft Project, Primavera and RPLan. Project planning and scheduling is essential for medium to large size projects and hence for all kinds of development projects. The project plan defines the tasks to be executed and their resource requirements.

Conventional project management systems do not allow to associate the tasks in a project plan with according parts of the product model. Furthermore, they support only the planning phase of the project but not the execution of the defined tasks. No client applications exist for the actual performers of the tasks. The enactment state of the

development process is not reflected in a project management system. Consequently, monitoring and control of a development project by using project management systems only is difficult, costly, and error-prone.

Workflow Management Systems. Workflow management has recently gained in importance in the different engineering domains. *Workflow management systems* [2] (see column (5) of Table 1) are used to support well-defined personal and collaborative processes. The workflow approach allows for a partial automation of processes, and the available technologies enable interoperability with other systems and applications in service oriented architectures. The usefulness of workflow support for development processes has been identified in both academia [3,4,5,6] and industry. Workflow support has been integrated into life cycle asset information systems for plant design [7,8] and integrated software development environments.

The home ground of workflow management systems, however, is in other domains such as business process management in administrations, banks, or insurance companies. In these domains, they constitute the industrial state of practice in process support. Therefore, it is not surprising that vendors and users of workflow management systems are looking for solutions which make workflow management systems applicable to dynamic development processes.

However, workflow management systems are not suitable for the management of whole development processes. Rather, they support only the enactment of structured subprocesses which may be defined in advance. Unlike project management systems, global project planning goes beyond the scope of workflow management systems. The tasks to be executed evolve during the course of the project. Many tasks cannot be planned until certain intermediate results of the development process are available, e.g., the flowsheet of a chemical plant or the design of a software architecture. It is not feasible to model a complete development process as one workflow due to the inherent uncertainties and dynamics. As a consequence, the workflow instances in the workflow management system are disconnected and not embedded into the context of the overall process.

Integrated Project and Workflow Management Systems. For these reasons, several research groups have investigated the opportunities of *integrating project management systems* with *workflow management systems* [3,4,5,6] (see column (2) of Table 1). In all of these approaches, project plans are used for global planning, and structured subprocesses are enacted with the help of workflow management systems. Furthermore, integration components couple these systems such that workflows may be represented in project plans.

However, the different integration approaches all face the problem that no information about the process enactment state is maintained in the respective project management system. Thus, the ability to monitor projects is severely limited. Furthermore, since products and data flows are not represented in project plans, product management is still beyond the scope of the integrated systems, and data flows between workflow instances have to be managed manually. Altogether, these restrictions call for an extended project management system (see next subsection).

1.2 Contributions

This paper makes several contributions regarding tool support for process management. The contributions can be divided into three parts which refer to the last three columns in Table 1.

Process Management System. The *A*daptable and *H*uman-Centered *E*nvironment for the M*A*nagement of *D*evelopment Processes (AHEAD) [9,10,11,12,13,14,15,16]) was developed within the long-term research project IMPROVE [1] and was applied to multiple domains (chemical, mechanical, and software engineering). AHEAD is a management system for development processes which provides integrated support for managing products, activities, and resources. A management system which offers this kind of integrated functionality is called a *process management system* throughout this paper (column (3) of Table 1).

AHEAD may be considered as an extended project management system. AHEAD differs from conventional project management systems inasmuch it manages the products of development processes in addition to activities and resources. Furthermore, it maintains state information of tasks and offers a work environment which developers use for performing the tasks assigned to them. In these ways, AHEAD integrates process planning and enactment, which offers great opportunities for the monitoring and control of development processes since the current execution state can be directly compared to the plan.

In contrast to AHEAD, workflow management systems do not support project planning. Their main contribution consists in the (partial) automation of routine processes. Furthermore, AHEAD differs from workflow management systems inasmuch it has been designed from the very beginning to allow for seamless interleaving of process planning and enactment. In contrast, most workflow management systems require a pre-defined workflow and support dynamic changes during enactment only to a limited extent.

Integrated Process and Workflow Management Systems. In the process management system AHEAD, activities are managed with the help of *dynamic task nets* [17]. The degree of automation of tasks is very limited on this level of process management. Work packages are assigned to developers who are responsible for delivering the specified results. For developers, a work environment is provided which maintains the documents required for each task and offers commands for activating tools operating on these documents. However, process automation as supported by workflow management systems is not provided.

Thus, process and workflow management systems offer complementary functionality which may be combined with the help of integration tools. As a result, we obtain an *integrated process and workflow management system* (column (4) of Table 1). This integration provides added value in particular when a company has already applied workflow management systems in a fragmentary way in its development projects. Then, the integrated system allows to reuse pre-defined workflows for process fragments, which are combined into a coherent development process.

Compared to the integration of project and workflow management systems, our solutions [15,18,19,20] provide for a tighter integration: Workflows are not only represented

in the task net managed by the process management system. In addition, the enactment state in the workflows may be monitored by transforming workflow operations onto state transitions in the task net. Furthermore, data flows may be propagated from the process management system to the workflow management system in order to provide workflows with inputs. Conversely, outputs created in the workflows are represented in the process management system such that they may be routed to successor tasks. Altogether, a much tighter integration is achieved than in systems integrating project and workflow management.

Adding Dynamics to Workflow Management Systems. State-of-the-art workflow management systems permit no or only limited deviations from the workflow definition at runtime of a workflow. However, even in the domains where workflow management systems are the state of practice for process support, such as business process management in administrations, banks, or insurance companies, the need for dynamic changes of workflows at runtime has been identified. For example, standard examples for routine processes such as travel reimbursement claims encompass so many alternative paths of execution that it is difficult to define a workflow which covers all of them. This argument is reinforced in the domain of development processes, which are less repetitive by their very nature. Therefore, using static workflows in dynamic processes alone does not provide a comprehensive solution to the problem of dynamic changes.

Many research projects focus on the development of *flexible workflow management systems* [21,22,23,24]. All prototypes have in common that they have been developed from the start to support dynamic workflows, i.e., flexibility is built *a priori* into the respective workflow engine. However, if a company has invested in a workflow management system which is widely used in its development projects, it may want to retain its investments and keep the system in use. This requires *a posteriori integration*: Tool support has to be developed in order to add dynamics to a legacy workflow management system.

Driven by this motivation, we have *added dynamics* to an existing *workflow management system* [25,26]. In this way, we have improved the capabilities of the workflow management system such that it may be applied to dynamic processes (contributing an improvement to column (5) of Table 1). In contrast to the flexible workflow management systems mentioned above, we had to solve the challenging problem of adding dynamics without modifying the given workflow management system. This problem was solved by a tool for dynamic changes which allows to add and delete workflow activities while the workflow is being enacted.

1.3 Structure of the Paper

The rest of this paper elaborates on the different and complementary support tools for dynamic development processes which have been described in Subsection 1.2. Section 2 presents the AHEAD process management system (column (3) of Table 1). Our solutions for using workflows in dynamic processes are described in Section 3 (column (4) of Table 1). Section 4 explains how we have added dynamics to a commercial workflow

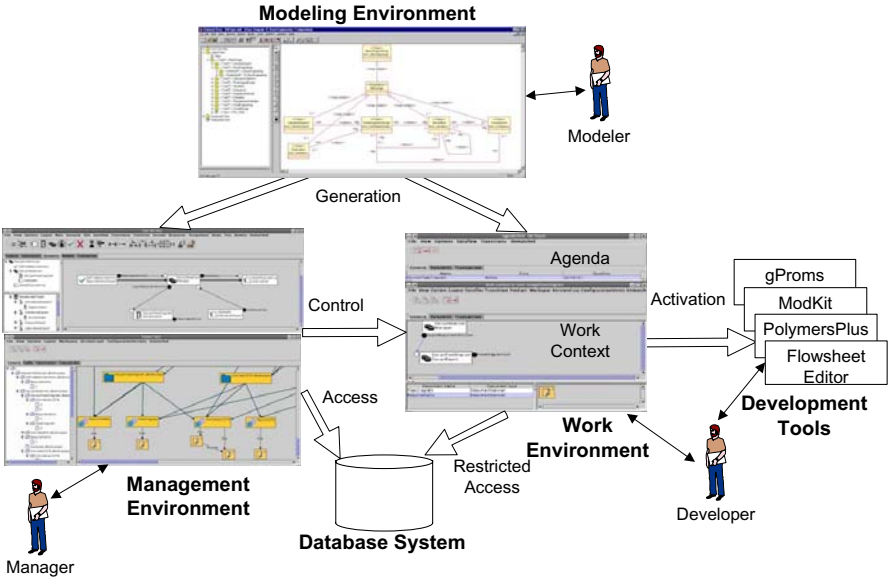


Fig. 1. Overview of the AHEAD system

management system (column (5) of Table 1). Related work is compared at the ends of all of these sections. The technical sections are followed by a discussion (Section 5). Finally, Section 6 concludes the paper.

2 The Process Management System AHEAD

The process management system AHEAD supports the integrated management of products, activities and resources for dynamic development processes. Subsection 2.1 provides an overview of the AHEAD system. Subsection 2.2 describes dynamic task nets for activity management, focusing on dynamic changes at run time. Subsection 2.3 explains how AHEAD may be adapted to a specific domain by defining a process model. Subsection 2.4 deals with deviations from and evolution of process models. A discussion of related work (Subsection 2.5) concludes this section.

2.1 System Overview

Figure 1 gives an overview of the AHEAD system. AHEAD offers environments for different kinds of users, which are called *modeler*, *manager*, and *developer*, respectively.

The *management environment* supports project managers in planning, analyzing, monitoring, and controlling development processes. It provides graphical tools which address the management of activities, products, and resources, respectively:

- For *activity management*, AHEAD offers dynamic task nets which allow for seamless interleaving of planning, analyzing, monitoring, and controlling. Dynamic task nets may be considered as extended project plans which in particular provide state information and include inputs and outputs of tasks as well as data flows in addition to control flows.
- *Product management* is concerned with the products of development processes, their versions and relationships (i.e., with the management of documents and models as supported by product/engineering data management systems, document management systems, or software configuration management systems).
- *Resource management* deals with the management of human resources (i.e., the members of the project team) which are assigned to development tasks, taking their roles and capabilities into account.

In the rest of this paper, we will focus on the management of activities, and we will not elaborate further on the management of products and resources.

AHEAD does not only support managers. In addition, it offers a *work environment* which consists of two major components:

- The *agenda tool* displays the tasks assigned to a developer in a table containing information about state, deadline, expected duration, etc. The developer may perform operations such as starting, suspending, finishing, or aborting a task.
- The *work context tool* manages the documents and tools required for executing a certain task. The developer is supplied with a workspace of versioned documents. He may work on a document by starting a tool such as e.g. a flowsheet editor, a simulation tool, etc.

The management environment and the work environment are both used at project run time. Both environments are highly *interactive*: The manager is provided with interactive tools for managing the development process, and the developer utilizes a work environment which provides commands for state transitions and tool invocations.

AHEAD consists of a generic kernel which is domain-independent. This means that the management environment and the work environment may be applied "out of the box". Optionally, models of development processes may be defined with the help of the *modeling environment*. For example, in order to apply AHEAD to design processes in chemical engineering, the modeler may define task types for flowsheet design, steady-state and dynamic simulation, etc. From a process model, code is generated for adapting the management and the work environment. The adapted system still provides seamless interleaving of planning and enactment, but offers pre-defined types for instantiating tasks, control flows, etc.

2.2 Dynamic Task Nets

A *dynamic task net* consists of tasks that are connected by hierarchical and non-hierarchical relationships:

- Tasks may be decomposed into subtasks, resulting in *task hierarchies*. Complex and atomic tasks are assigned to managers and developers, respectively.

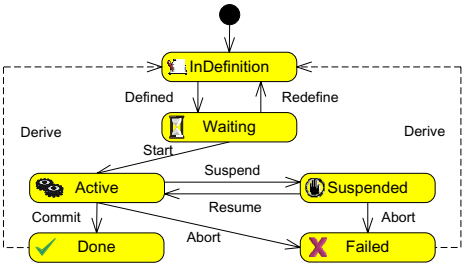


Fig. 2. State diagram

- *Control flows* resemble precedence relationships in Gantt diagrams. A sequential control flow corresponds to an end-start dependency. A simultaneous control flow enforces both start-start and end-end dependencies. Finally, a standard control flow represents an end-end dependency.
- *Feedback flows* are oriented oppositely to control flows. They are used to represent feedback in the development process, which is not possible in a conventional project plan.
- Each task has *inputs* and *outputs* which may be considered as ports for consuming and producing the products of development processes. *Data flows* describe the routing of documents along horizontal and vertical task relationships.

Seamless interleaving of planning and enactment constitutes the core mechanism for supporting dynamic changes in the AHEAD system at project run time. The manager may modify the task net at any time in the course of the project. Only minimal constraints are imposed on these modifications based on the state of execution. For example, it is not allowed to delete an active task because otherwise work performed by the developer would be lost. Since tasks are performed by humans (with the help of development tools), changes may be accommodated more easily than in the case of automated processes. For example, a new input may be attached to a task which has already started; it is up to the assigned developer how to process this input. In the case of an automated process, a change of this type usually has to be prohibited.

Thus, at project run time both edit operations and enactment operations may be performed on a task net. *Edit operations* change the structure of the task net. *Enactment operations* change the states of tasks (see below) or produce or consume data. All of these operations have to take the current enactment state into account.

The enactment states of tasks and their allowed changes are defined by a *state diagram* (Figure 2). In the initial state InDefinition, only edit operations are permitted on the current task (creation of input and output ports, creation of a refining task net in the case of a complex task). In Waiting, the task waits for its activation. In the state Active, both edit and enactment operations are allowed. When a task is Suspended, it may no longer produce or consume data, and it may have no active subtasks; editing operations are still allowed. Successful and failing terminations are represented by the states Done and Failed, respectively. In these states, no changes are allowed at all to ensure traceability of the development process.

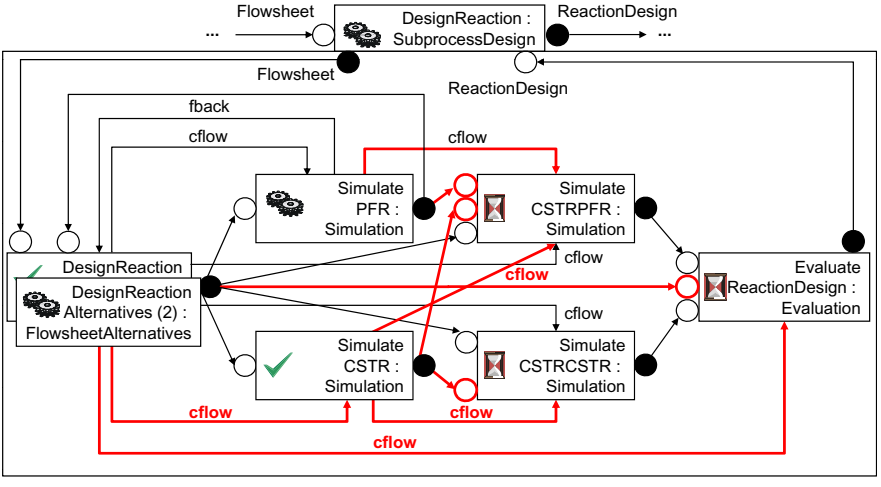


Fig. 3. Dynamic task net

Traceability is further supported by different ways of *version control* for elements of dynamic task nets:

- If a terminated task has to be reactivated, a new task version is created via the Derive operation[1]. The data of the old version remain unaffected and are copied to the new version.
- While a task is active, it may produce and receive multiple versions of outputs and inputs, respectively. Therefore, the tokens referencing these product versions are versioned, as well.

Figure 3 shows a snapshot of a (cutout of) a task net for a design process from the chemical engineering domain[2]. In the diagram, each task is represented by a rectangle containing its name, its type, and its state (displayed as an icon). Control, feedback and data flows are shown as arrows, composition relationships are represented by nesting component tasks into a box for the composite task.

In the sample process, which deals with the design of the reaction part of a chemical process, the designer has inserted four alternatives into the flowsheet: using single reactors of different types (CSTR and PFR, respectively) and using reactor cascades (CSTR-PFR and CSTR-CSTR, respectively). The designer already knows that eventually one of the cascades will be selected as the best alternative. However, simulating these cascades in a single step is too complex. Therefore, the individual reactors are simulated first, and the simulation results are propagated to the tasks for simulating the cascades.

[1] Strictly speaking, Derive is not a transition since it involves the creation of a new object (a new task version) rather than the state change of an existing object.

[2] The elements displayed with red thick lines denote inconsistencies with respect to the process model definition; see next subsection.

The task net of Figure 3 illustrates a state of enactment which is reached at some time during the course of the respective development project. In particular, the task net depends on the reaction alternatives, which are elaborated only in the initial design task. Even when these alternatives are known, it is not possible to generate the task net from the product structure. As explained above, the reaction alternatives are investigated in a certain order which has to be determined by the manager. Furthermore, unanticipated feedback may occur at any time, requiring changes to the task net for the purpose of feedback processing.

In the current state of enactment as shown in Figure 3, the tasks for simulating the reactor cascades as well as the final evaluation task still reside in state Waiting. The task for simulating the CSTR was already completed successfully. In contrast, the designer who is responsible for simulating the PFR detected a problem which caused the creation of a feedback flow to the initial design task. Since this task had already been committed, a new task version (rectangle in front) had to be created which now resides in state Active.

2.3 Process Model Definitions

The AHEAD system may be applied "out of the box" using so-called standard types for tasks, control flows, data flows, etc. Optionally, a domain-specific *process model definition* may be created which introduces domain-specific types and constraints. Processes are defined with the help of UML diagrams which have been tailored to process modeling through a UML profile defining adapted versions of class, communication, and state diagrams using stereotypes and tagged values [27, 28, 14].

With the help of *class diagrams*, process models may be defined at the type level. Figure 4 shows a class diagram taken from the reference scenario developed and used throughout the IMPROVE project [29]. This scenario deals with the basic engineering of chemical plants. For the task class interface SubprocessDesign, a realization class is defined which involves simulations of the respective chemical subprocess (as an alternative to laboratory experiments[3]) . For the simulation-based realization, a refining task net is defined on the type level. This task net consists of exactly one task for defining alternative chemical processes (class FlowsheetAlternatives), at least one Simulation task, and exactly one Evaluation task which selects the best alternative and delivers the overall subprocess design to the parent task. Tasks of these classes are ordered by control flow associations. Furthermore, a feedback flow association is defined from Simulation back to FlowsheetAlternatives. As in instance-level task nets, outputs and inputs are represented by black and white circles, respectively. Finally, data flow associations are used to connect outputs to inputs along vertical or horizontal task associations.

Class diagrams primarily serve to define *structural models*. Structural elements may be augmented with behavioral properties using tagged values (as e.g. EnactmentOrder = simultaneous in Figure 4). Furthermore, state diagrams and communication diagrams are offered for *behavioral modeling* (which are not described further here).

[3] In the process model definition, multiple realizations may be defined for a single interface. However, on the instance level the (single) realization of a task has to be selected when the task is instantiated. Furthermore, interface and realization are merged into a single object.

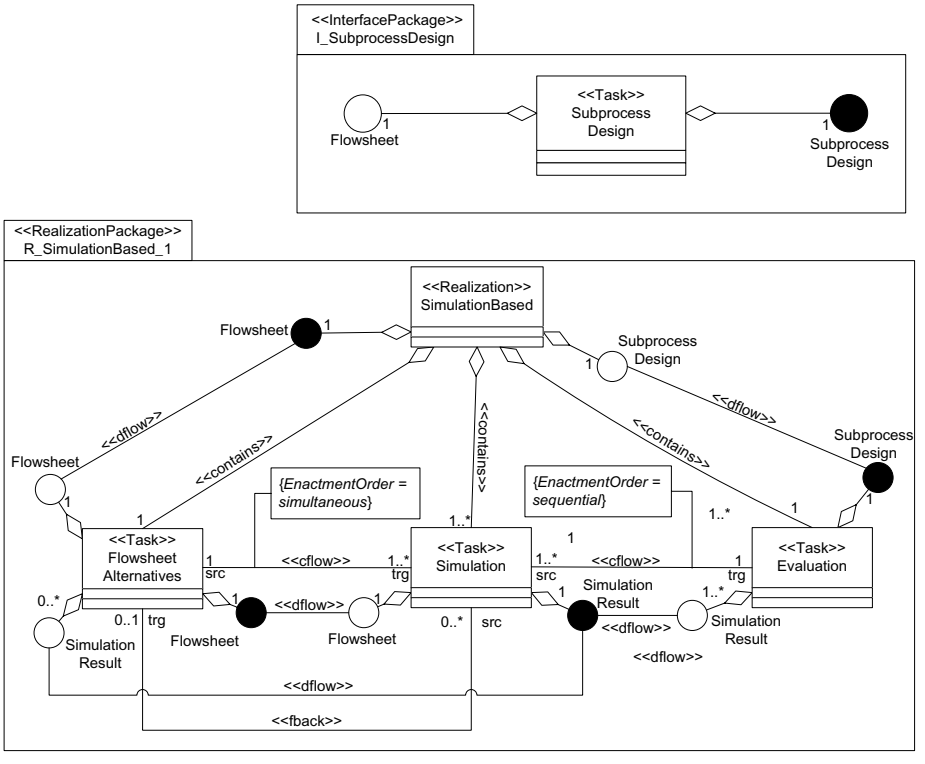


Fig. 4. Class diagram as process definition at the type level

Altogether, AHEAD provides a *wide spectrum approach* to process modeling. The spectrum covers ad hoc processes, which are based on standard types rather than on a domain-specific process model definition, process models defined on the type level with the help of class diagrams, and instance-level process patterns defined by communication diagrams.

2.4 Process Model Evolution

Despite of the inherent flexibility of our object-oriented approach to process modeling, it may happen that a process has to deviate from its definition. For example, the process model definition may not include the class of some task which has to be performed in the project at hand, or a control flow which is defined as sequential prohibits the overlapping enactment of the source and the target task. Therefore, AHEAD allows for *controlled deviations* of a task net instance from its definition: The project manager is supplied with commands for allowing/disallowing deviations at the level of subnets of the overall hierarchy.

Deviations result in *inconsistencies* which are signaled to the manager [14]. In Figure 3, inconsistent elements are emphasized with bold face fonts and thick lines in red

color. For example, the control flows between simulation tasks are marked as *structurally inconsistent* since they were not anticipated by the process modeler. Furthermore, the control flow ending at the task SimulateCSTR is *behaviorally inconsistent*: The task has already been terminated while its (reactivated) predecessor is still active.

Deviations may trigger process model evolution. In our example, the structural inconsistencies may be removed by adding further elements to the process model which were missing so far. This can be achieved by creating a new *process model version* and *migrating* the process model instance to the new version. In this way, AHEAD supports *round-trip process model evolution*.

2.5 Related Work

By using class diagrams, we follow an *object-oriented approach* to process modeling. This approach differs from the *procedural approaches* realized in workflow management systems [2,30,31,32]. The object-oriented approach provides for more flexibility since a task net may be composed specifically for the project at hand at run time by dynamic instantiation of task classes and associations. In contrast, instantiation of a workflow means that a pre-defined workflow is started.

AHEAD may rather be considered as an *extended project management system* than as an extended workflow management system. In particular, the *control flows* available in AHEAD strongly resemble precedence relationships in project plans. Unlike workflow management systems, AHEAD does not support control structures involving conditions. It was a deliberate decision not to support these control structures because we considered it too demanding for the project manager to develop a plan with multiple variants.

A variety of mechanisms has been designed and implemented for supporting dynamic processes (not only in workflow management systems, but also in *process-centered software engineering environments* [33,34]). In the following, we will list some of these mechanisms (without striving for completeness):

Ad Hoc Processes. Processes are defined on the fly for each specific case, i.e., there is no predefined process definition [21].

Exception Handling. Processes defined for the "standard case" of processing are extended with exception handlers, which describe how to react on deviations from the standard case [35].

Flexible Control Flows. Many process definitions suffer from imposing too rigid constraints on the order in which process steps are executed. Thus, definition languages have been developed for describing more flexible control flows [22,36].

Late Binding. Process definitions are usually decomposed into subprocesses to manage complexity. In the case of late binding [37], the definition of a subprocess may be deferred until it is going to be executed.

Interleaving of Definition and Enactment. A more general approach to dynamic changes supports seamless interleaving of definition and enactment in a similar way as in integrated programming environments, where a program may be modified during execution [38,39,23].

Toleration of Inconsistencies. Flexibility is increased by tolerating inconsistencies of instances with respect to their definitions [40]. For example, in [41] a process step may be executed even if its preconditions are not satisfied.

Version Control for Process Definitions and Migration. In these systems, process definitions are put under version control [24, 42, 43]. Furthermore, running instances may be migrated to revised definitions.

AHEAD supports *dynamic changes* through ad hoc processes ("untyped" task nets), interleaving of planning and enactment, and toleration of inconsistencies. Furthermore, *process evolution* is supported through version control of process definitions and migration of process instances. AHEAD does not address exception handling and flexible control flows, which are partially obsolete since process instances may be changed flexibly at run time. Moreover, AHEAD does not support late binding.

3 Using Workflows in Dynamic Processes

In this section, the integration of a workflow management system with the management environment for development processes in AHEAD is described. This integration is beneficial when existing workflow management systems within an organization and all existing workflows can be utilized to represent relevant workflow details within the overall development process. Not all details modeled in workflows have a real value for the process manager on the abstract process planning level that addresses the coordination and distribution of work in the managed overall development process. Instead, only relevant aspects of workflow processes need to be monitored by the manager. For this purpose, we have developed a generic integration approach which is applicable to the majority of integration scenarios and deals with the desired partial monitoring of workflows in development processes. With respect to the details of the integration, several alternatives can be distinguished which are more or less appropriate in certain scenarios.

Subsection 3.1 briefly explains the motivation for our work. In Subsection 3.2 the coupling of a workflow management system with the AHEAD system is described so that workflows can be projected into dynamic task nets and monitored by the process manager in the management environment of AHEAD. Subsection 3.3 describes the weaving of workflow processes with other (dynamic) process parts within the overall dynamic process and Subsection 3.4 explains how workflow modeling capabilities can be additionally used. Finally, Subsection 3.5 gives an account of related work.

3.1 Motivation

We have developed an approach to integrating workflow processes into the overall development process and have realized a coupling of workflow management systems with AHEAD for use within an organization. In this setting, the AHEAD system is used within the organization as the central instance for the project planning of the development process and the coordination of all (human) work that is carried out during the process. The overall dynamic development process is represented as a dynamic task

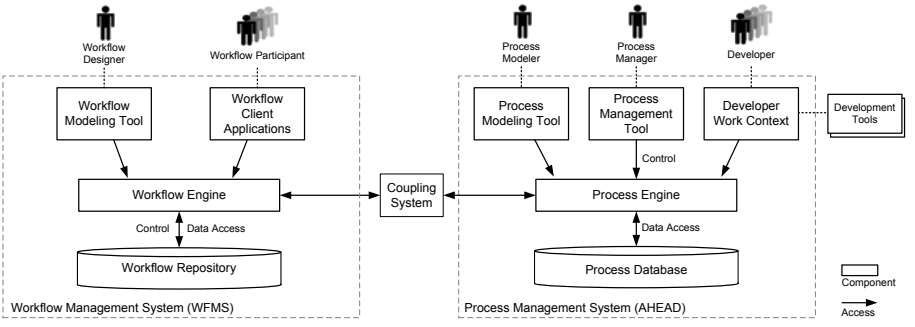


Fig. 5. Coupling of a workflow management system with the process management system AHEAD

net and can be *monitored* and controlled by the process manager within the management environment of AHEAD. Repetitive routine processes are executed in workflow management systems. The workflow processes are integrated as *process views* [15,19] into the overall dynamic processes and projected onto dynamic task nets. The process manager can monitor all parts of the development process within the management environment in the same process notation of dynamic task nets. In this way, an overall development process comprising dynamic and repetitive partial process fragments is executed across heterogeneous process support systems.

The outlined integration approach is based on the observation that although development processes are generally human-driven, some fragments of the development processes are indeed of a repetitive nature, i.e., repetitive process definitions can be defined for these fragments and modeled as workflow definitions. A similar approach to use workflow management systems within human-driven development processes is described e.g. in [44].

Within a prototypical realization, the AHEAD system has been integrated with the workflow management system SHARK from ENHYDRA [45]. A full account of the coupling approach and the realized prototype is given in [15,18,19]. The resulting solution approach was adopted in the technology transfer project T6 [19,20] for the integration of the *Proc*ess Management *E*nvironment for *E*ngineering *D*esign Processes (PROCEED) with a workflow management system. PROCEED is a new implementation of AHEAD based on an industrial platform.

3.2 Monitoring Workflows in Dynamic Processes

Figure 5 shows the resulting architecture for a coupling of a workflow management system (left) with AHEAD (right) via an event exchange infrastructure. The AHEAD system triggers the instantiation of a workflow. Upon the triggering event, the workflow management system spawns a new workflow instance and creates new run time instances for the first startable workflow activities of that workflow definition. All necessary input data is transferred from AHEAD to the workflow management system at this moment for further consumption within the new workflow instance. The created

workflow-activity instance is immediately started in the workflow management system to begin the workflow execution. In particular, for each human workflow activity, a developer can start working on this activity. During task execution, all relevant workflow changes (activity or task state changes, resource assignment changes, data changes) are signaled as events to the AHEAD system where the corresponding task net that represents the workflow process is updated accordingly. Vice versa, all necessary changes in the workflow (and state changes) are signaled from the AHEAD system to the workflow management system as events where they are processed in a similar way.

Mapping of Workflow Processes into Dynamic Task Nets. Workflow processes can be represented within the overall dynamic task net via a *projection* of the workflow process onto a dynamic task net fragment (called *workflow task net fragment*). Each task in a workflow task net fragment that represents a workflow activity is called a *workflow task*.

The transformation of workflow processes to dynamic task nets does not need to preserve the full semantics of both formalisms, because this would lead to very rigid requirements for the systems to be integrated. Since the workflow fragments are used in AHEAD for monitoring purposes only, it is tolerable if some process information is lost during the generation of the workflow fragments. Thus, only selected details of the workflow process, which are necessary to represent the coordination aspects of the activities in the workflow, are mapped into a dynamic task net. The projection is realized with a transformation algorithm to convert a workflow process definition into a corresponding dynamic task net fragment definition. While a brief summary is given in the sequel, more details of the chosen transformation projection and the coupling infrastructure are described in [46].

Within AHEAD, a workflow task net fragment can then be inserted into the task net at a location that is selected by the process manager (i.e. as children of the parent task in the current process fragment). Initially, it is isolated at this moment from the rest of the dynamic task net. Hence, the process manager has to connect tasks from the new imported workflow task net fragment with tasks from the remaining dynamic task net via control flows and data flows (a *weaving* step).

In order to reduce the effort for the integration of multiple different workflow management systems, we make use of a neutral exchange format to represent a wide-spread workflow modeling language. The language XPDL has been chosen since it is a standard notation brought forward by the Workflow Management Coalition [31]. Therefore, multiple workflow management system products that adopt this standard can be supported with the coupling solution in general.

The projection defines both the structural mapping of workflow structures onto task net structures as well as the mapping of the state diagram of workflow activities (in XPDL) onto the state diagram of AHEAD tasks. Additionally, the data flow between workflow activities is projected onto the data flow between AHEAD tasks. Workflow definitions can include subprocess calls and iterations. These mapping details are explained in the sequel.

To illustrate our projection approach, Figure 6 introduces a simple example of a workflow from the chemical engineering domain and shows its projection into a dynamic task net fragment. The underlying scenario in the chemical engineering domain

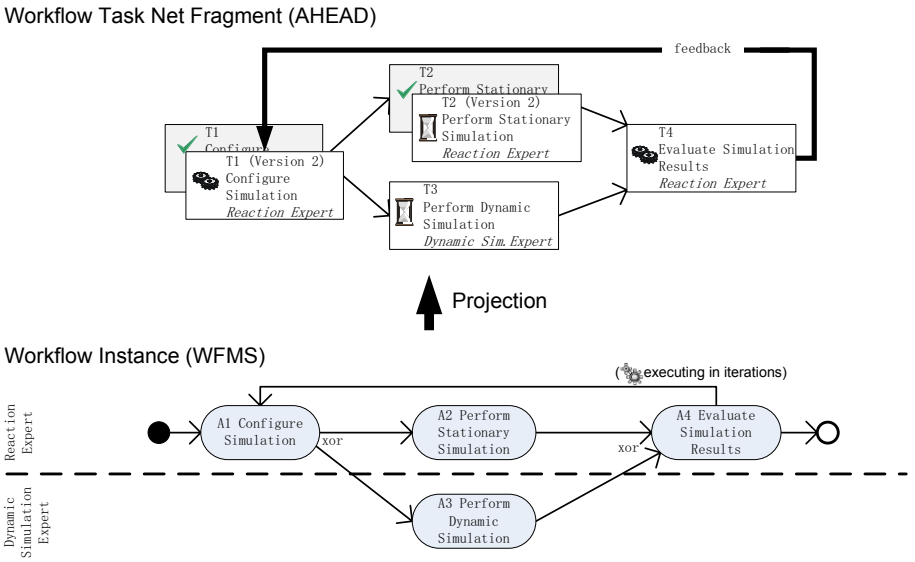


Fig. 6. Workflow instance and according task net fragment

has already been introduced in Section 2. The design of a certain part of the chemical plant during the development process, such as the design of the reaction part, can be supported with workflow management system technology. Specific routine processes can be executed within workflow management systems but need to be monitored within dynamic task nets as the contained process steps are important on the coordination and management level for the process manager.

The example represents a routine subprocess for performing simulation calculations for a given reactor part in the flowsheet. This process refines the simulation task defined in Figure 4 and instantiated multiple times in the task net of Figure 3. Simulations are carried out iteratively until a satisfactory result is achieved. Two different simulation methods can be used here to perform the simulation: A stationary or steady-state simulation allows to simulate the reactor with respect to the equilibrium state while a more complex (and more expensive) dynamic simulation investigates the behavior of the chemical process in non-steady states (e.g., start of the reaction or technical faults). For this scenario, the lower part of the figure shows the workflow process (workflow management system) and the upper part shows the projected dynamic task net (AHEAD). The workflow is comprised of four activities: In a first activity A1 Configure Simulation, a reaction expert estimates the requirements for the desired simulation and decides which type of simulation is carried out. According to this decision, in each iteration either activity A2 Perform Stationary Simulation is executed by the reaction expert or A3 Perform Dynamic Simulation is delegated to a simulation expert with specific expert knowledge (only one of both activities is executed). In activity A4 Evaluate Simulation Results, the reaction expert interprets the simulation results and decides if the simulation data are sufficient or if the simulation has to be repeated with modified

Table 2. Mapping of workflows to dynamic task nets [15]

Workflow Model Concept(XPDL)		*Process Model Concept(AHEAD)*
Workflow Process	↔	Task Net
Workflow Activity	↔	Task
Workflow Participant	↔	Performer
Data Field	→	Document
Formal Parameter	→	Document
Transition Information	↔	Control Flow (seq.)

configuration parameters or a different simulation method (iteration from activity A4 to A1). This example is used throughout the remainder of this subsection.

Structural Mapping. The mapping of the most important elements of the workflow meta model in XPDL to elements of the AHEAD meta model is described by a set of mapping rules as follows (summarized in Table 2). A workflow process definition from XPDL is mapped to a task net in AHEAD. All activities in a workflow process definition are mapped to AHEAD tasks. In the example of Figure 6, activities A1-A4 are mapped to tasks T1-T4 in the dynamic task net. All tasks are inserted at the same time into the development process. However, for an activity there can be multiple task versions in the task net (due to iterations, see below). This is shown in the figure for activities A1 and A2 which have two different task versions in the task net. For subprocess definition activities and route activities in XPDL, a new task is contained in the task net. All workflow activities within an XPDL subprocess are mapped to AHEAD tasks likewise.

Workflow participants (XPDL) are assigned to tasks in the task net and assignments of participants to workflow activities are carried over to AHEAD by assigning participants to tasks. In Figure 6, the workflow participants Reaction Expert and Dynamic Simulation Expert are mapped to corresponding resources in the dynamic task net. For example, the assignment of activity A1 to participant Reaction Expert from the workflow process is also visible in the task net where task T1 is assigned to the corresponding resource (shown in the bottom line of the task's label).

For workflow-relevant data from XPDL, corresponding document elements are created in AHEAD (not shown in the figure). Updates to workflow-relevant data (in workflow variables) are mapped to the creation of new versions of the corresponding documents. Not all workflow-relevant data (variables) need to be mapped to AHEAD task nets. For pragmatic reasons, a naming scheme was defined to automatically separate mapped from unmapped workflow-relevant data.

Transitions, Iterations, and Feedback Flows. Transitions between workflow activities in XPDL are transformed into sequential control flows in AHEAD (as shown in Figure 6). Please note that XOR-transitions and AND-transitions are both mapped in the same way[4]. The corresponding state machine mapping is described below. A workflow activity inside a loop control structure may be associated with multiple task versions

[4] The mapping is not injective, i.e., the transformation implies a loss of information. As mentioned earlier, branches cannot be represented in AHEAD. AND- and XOR-transitions may be distinguished in AHEAD only by monitoring the enactment of the respective workflow.

Table 3. State mapping between workflow activities and tasks

Workflow Activity State(XPDL)		*Process Task State(AHEAD)*
—	↔	InDefinition
open.not_running.not_started	↔	Waiting
open.not_running.suspended	↔	Suspended
open.running	↔	Active
closed.completed	↔	Done
closed.terminated	→	Failed
closed.aborted	→	Failed

(in the task net) according to the iterations of the loop. For example, the activity A1 in the workflow is associated with two versions of task T1 in Figure 6. While the loop in the workflow process expresses that the activities A1–A3 have to be repeated until a success criterion matches, in the task net the concept of *feedback flow* is used to express the same meaning. In our example, initial versions of tasks T1 and T2 are executed. Subsequently, T4 detects a failure. To repair the failure, a second version of task T1 is instantiated and a feedback flow from T4 to the new version of T1 is created. Subsequently, the new version of T1 is reactivated. Since the steady-state simulation has to be performed once more, a new version of T2 is created and prepared for enactment.

Dynamic Semantics: State Machine Mapping. Table 3 shows the mapping of the state machine of tasks in a dynamic task net, workflow instances and workflow activities. The coupling is described here informally. When a workflow activity is in a pre-activation state (open.not_running.not_started) so is the corresponding AHEAD task (here: Waiting). The task state InDefinition in AHEAD is only used within AHEAD and has no direct mapping from a XPDL state. When a workflow activity enters the running state (open.running), the corresponding AHEAD task is activated (Active). The XPDL meta model defines three terminal states, namely successful termination (closed.completed), user-initiated cancellation (closed.terminated), and system-initiated cancellation (closed.aborted). The AHEAD meta model defines only two terminal states for successful termination (Done) and failed task cancellation (Failed). The workflow state closed.completed is mapped to task state Done, while the additional information provided in XPDL is lost during the mapping when both cancellation states closed.terminated and closed.aborted from XPDL are mapped to the cancellation state Failed in AHEAD.

3.3 Weaving of a Workflow Fragment into a Dynamic Task Net

The individual workflow processes which are enacted by means of the workflow management system are all part of one overall development process which is represented by a hierarchically structured task net in the process management system. Therefore, the workflow fragments representing the workflow instances have to be embedded into the task net.

The workflow activities are represented by workflow tasks in the workflow fragment. The workflow instance itself can also be represented by a task in the task net. In this

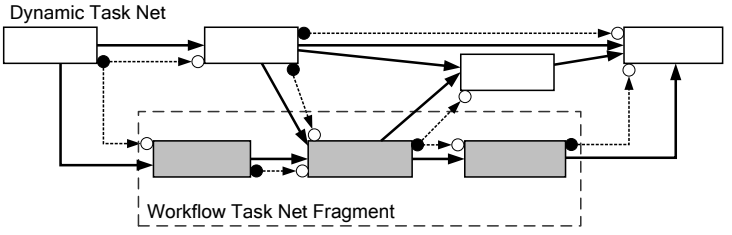


Fig. 7. Weaving of a workflow fragment into a task net

case, the subtasks of this task represent the workflow activities and no other tasks are contained in its realization.

However, the workflow instance does not necessarily have to be represented by a task in the task net. Alternatively, the tasks representing the workflow activities can be directly embedded into an existing task net. In this case, there may be sibling tasks which do not represent workflow activities.

When a subprocess of a development process is enacted in a workflow management system, the enactment of the workflow may depend on the state of tasks which are not part of the workflow. This is mostly the case because data or documents which are created outside of the workflow are required in certain activities of the workflow, e.g. when a workflow for the permit procedure of a chemical plant requires at a certain step the result of the cost calculation. On the other hand, the enactment of the workflow and its (intermediate) results may influence other tasks in the development process, e.g. when a workflow is enacted for the specification of a device, and this specification is required for the procurement of the device later on. Altogether, two different cases for the embedding of a workflow into the surrounding development process may be distinguished.

In the first case, the workflow as a whole requires input data from preceding tasks and produces results which are required in succeeding tasks. All required inputs have to be available at the start of the workflow and cannot be provided later at run time. Intermediate results of the workflow are not available in the surrounding process. This case requires that the workflow instance is represented by a task in the dynamic task net.

In the second case, the individual activities of the workflow may consume external data from outside the workflow and may provide intermediate results which can be used by external tasks. This is illustrated in Figure 7, where the gray boxes surrounded by the dashed line represent workflow tasks in the workflow task net fragment while the white boxes represent external tasks in the surrounding process. Required inputs for workflow tasks can be provided by external tasks as required, even after the start of the workflow, and intermediate results of the workflow are available for external tasks before the completion of the workflow. This case in which individual workflow tasks are connected with external tasks is called *fine grained weaving* of processes. In this case, it is not required that the workflow instance is represented by a task in the dynamic task net. Data inputs and outputs do not have to be defined for the workflow instance but can be directly defined for workflow activities.

The weaving of a workflow with the surrounding process imposes several technical requirements for the integration of the respective management systems. The handover of data between the two systems requires *well-defined data formats* which can be written and read by both systems. In case of documents which serve as input for tasks or workflow activities it may be required to transfer uniform resource locators (URLs) which specify the location of the respective documents in a common repository.

In the case of fine grained weaving, control flow connections between the tasks in a workflow fragment and the surrounding process context influence the execution of the workflow and the surrounding process. In the case of outgoing control flows, the workflow management system should provide an interface for the notification of external systems about state changes of activities and workflows. If such an interface is not available, a pull mechanism could be realized in which the process management system queries the workflow management system for changes in regular intervals. Vice versa, the execution of a workflow instance should respect the constraints imposed by incoming control flows. Different technical solutions are possible. The original workflow definition could be augmented by additional activities and control flows which represent the context of the workflow. Alternatively, special workflow activities and according workflow variables could be used in the original workflow definition to allow for later definitions of control flows from external tasks.

3.4 Workflow-Managed Dynamic Task Nets

The main motivation for the integration of AHEAD with a workflow management system is the continued utilization of existing workflow management solutions. In this case, the process participants who used the workflow management system before the integration, will go on using the client applications of the workflow management system after the integration. In particular they will use the worklist handler of the workflow management system to get informed about their assigned tasks.

The workflow fragment which is embedded into the dynamic task net of the overall development process serves the project manager mainly to view the status of the workflow tasks. The workflow tasks in the fragment may be atomic, i.e. they are not refined by any subtasks, or the tasks may themselves represent workflow instances and can therefore be refined by workflow fragments.

The second case accounts for the common scenario in workflow management system where one workflow invokes another workflow in a synchronous fashion, i.e., a workflow activity initiates the execution of another workflow and waits for its termination. This can be modeled in the dynamic task net by a task which represents at the same time the workflow activity and the invoked workflow instance. The subtasks of this task represent the activities of the invoked workflow.

Another motivation for the integration of an AHEAD-like process management system with a workflow management system may be to use the modeling capabilities for workflows, in particular alternative branching and loop structures, to enable a (partial) automation of the defined processes. This was the main motivation for the integration of PROCEED with a workflow management system in the technology transfer project T6 [19,20]. Workflows are not only monitored in PROCEED but the workflow fragments are also managed automatically by the workflow engine according to the enactment of

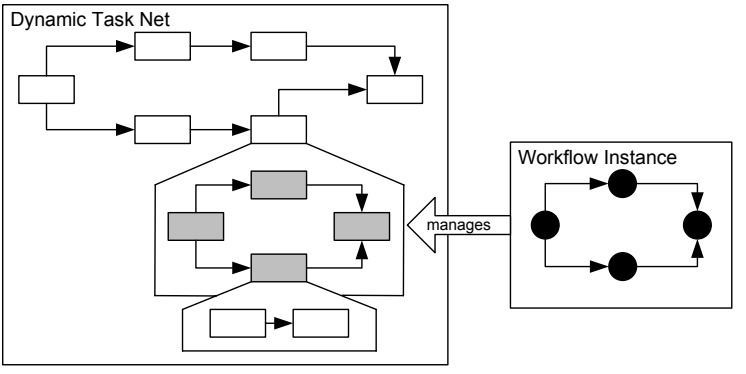


Fig. 8. Nesting of workflow-managed and manually managed dynamic task nets

the according workflow instances (tasks are prepared for execution, new task versions are created, etc.). In this case, the workflow task net fragments are called *workflow-managed dynamic task nets*. The advantage of this kind of integration is the usage of the workflow modeling capabilities and the workflow engine of the workflow management system. The process participants can manage their assigned tasks by means of the PROCEED work environment and do not need to use the worklist handler of the workflow management system.

The usage of the PROCEED work environment by the process participants has an influence on the required state machine mapping of PROCEED and the workflow management system. The start of a workflow activity has to be mapped to the transition of the according task in the workflow fragment from *InDefinition* to *Waiting* because tasks have to be started by the assigned resources and should not be activated by the workflow management system automatically.

To allow for the use of a workflow definition for a high-level subprocess of the overall development process, it is possible to refine workflow tasks in a workflow-managed task net by manually managed dynamic task nets. Otherwise, all subprocesses of the workflow process would have to be defined as workflows as well, which is infeasible for dynamic development processes. Figure 8 shows a hierarchically structured dynamic task net in which a subprocess is managed by the workflow engine but a workflow task is refined by a manually managed task net.

The management of task net fragments in PROCEED by a workflow engine imposes additional technical requirements for the integration of the two systems. A workflow activity whose corresponding task in the workflow fragment is realized by a manually managed task net has to wait for the successful termination of this task net in PROCEED before it can be terminated. This can be realized by the implementation of special workflow activities which wait for an external event before they terminate. Some workflow management systems provide this functionality.

3.5 Related Work

Several research groups have investigated the possibilities to integrate project management systems with workflow management systems.

Bauer distinguishes two approaches in [3]: loose coupling, where several workflow instances can be mapped to a single project task, and close coupling, where there is a one-to-one mapping of workflow activities and tasks in the project plan. The close coupling approach is not applicable, when the project plan and the workflows are on different abstraction levels. Hence Bauer presents a generic integration architecture for loose coupling which uses event-condition-action(ECA)-rules for data propagation and aggregation.

The MILOS tool [4] is an integrated solution for the management of software development processes. MS Project is used as a planning interface and is extended by means to define information flow between different tasks. The run time coupling between the workflow management component and MS Project is realized by means of ECA-rules.

In [5], the IPPM system is described which integrates features for both project and workflow management. An approach for the unfolding of workflow control structures to tasks in a project plan is presented.

In [6], Bussler discusses issues regarding the integration of workflow management systems and project management systems in general. Two approaches are distinguished for the mapping of control structures. Continuous mapping describes the case where tasks are inserted into the project plan at workflow run time according to the decisions made. With static mapping, all alternative paths are inserted before the start of the workflow.

The integration of AHEAD with a workflow management system is based on a well-defined mapping between the according process meta-models. Therefore, the definition of ECA-rules is not necessary, neither by the system developer nor by the user.

The different related integration approaches all face the problem that no information about the process enactment state and the data flow is available in the respective project management systems. As a consequence, fine grained weaving of a workflow instance with other tasks in the project plan is not possible. The context of a workflow instance in a project is merely defined by the complex task to which it belongs.

4 Adding Dynamics to Workflow Management Systems

4.1 Motivation

In the previous section, we were concerned with using workflows in dynamic development processes. By integrating process and workflow management systems, workflows are glued together in order to form a coherent development process. However, this approach does not release the restrictions of the used workflow management systems concerning support of dynamic changes.

In this section, we will present tools for *adding dynamics* to a *workflow management system* a posteriori. These tools were developed in a project in the business process domain. In contrast to the integration approach of the previous section, the extended workflow management system may be used as a stand-alone component. The primary goal was to make classical workflow applications more flexible. However, at the same

time this flexibility makes the workflow management system better suited for the application in development processes, which generally require more flexible support for dynamic processes than business processes. Therefore, the extended workflow management system may be integrated with a process management system along the lines of Section 3. In this way, the approach presented below complements the work described in the preceding sections.

In the AHEAD system, dynamic changes are supported by seamless interleaving of planning and enactment. In contrast, dynamic changes to repetitive processes are better handled by *deviation* from a workflow definition rather than by continuous evolution of some dynamic task net. Yet, classical workflow management systems do not allow for dynamic changes. In the following, we describe how and to which extent a classical workflow management system can be *upgraded a posteriori* in order to support even dynamic business processes. In particular, we extended the workflow management system *IBM WebSphere Process Server (WPS)*, which provides execution support for workflow definitions. In contrast to the concepts of Section 3, we just consider a single modeling language for executable models, namely the standardized language WS-BPEL [32] used in WPS.

In Subsection 4.2, we draw the big picture of the a posteriori extension of the workflow management system WPS before we elaborate on the kinds of dynamic changes which are actually supported by the approach (cf. Subsection 4.3). Finally, related work is discussed in Subsection 4.4.

4.2 Simulating Dynamics

Adding functionality for dynamic run time changes a posteriori to an existing workflow management system like WPS does not work in a straight forward fashion. WPS is a closed source system. Thus, adding dynamics by modifying the existing source was not an option. Instead, an additional architectural layer – called *dynamics layer* – was introduced (cf. Figure 9). This dynamics layer simulates dynamic changes inasmuch as workflow participants experience dynamic structural changes in their running workflow instances while the actual workflow definitions within WPS (forcedly) remain structurally unchanged.

The dynamics layer reflects the intrinsic distinction between build time and run time of classic workflow management systems as it consists of two parts. Existing workflow definitions are automatically augmented by a WS-BPEL transformer at build time by additional control flow structures. At run time, these additional control flow structures are used to deviate from the original control flow definition. Their run time behavior is controlled by the dynamics component which is the run time counterpart of the WS-BPEL-transformer. The additional complexities that come along with the dynamics layer are completely hidden from process participants. In the so called process model editor, which serves as a front end for process participants, dynamic changes are indeed displayed as structural changes in process instance models, i.e., models of running processes.

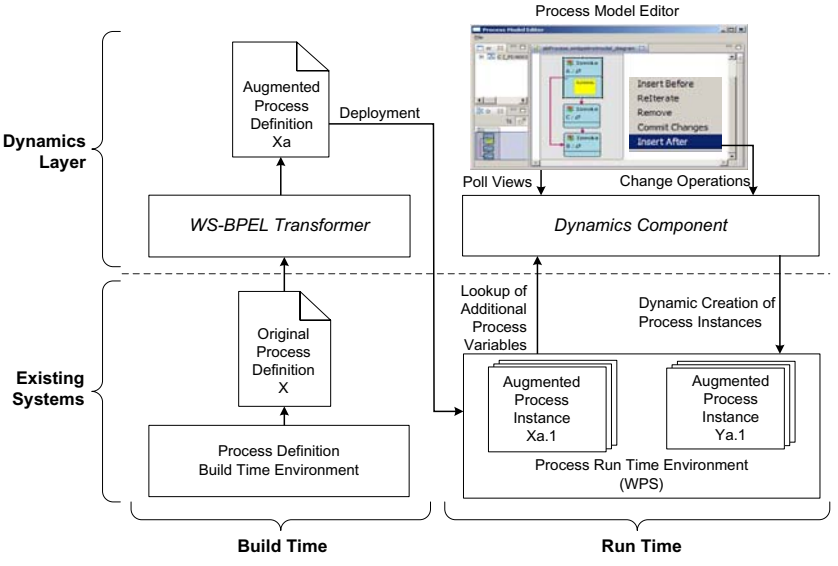


Fig. 9. Deviation of workflow enactment from workflow definitions

4.3 Dynamics Patterns

The realization of the dynamics layer is aligned with typical kinds of dynamic changes. These are called *dynamic patterns* which are described in detail in [25,26] and briefly in the following:

Dynamic Adding. Frequently, there is a need to dynamically add activities or even workflow fragments consisting of several activities to a running workflow instance. In a classical, i.e. static, workflow management system one could model placeholder activities — serving as *extension points* — among the actual activities. These added activities can then be used at run time for intermediate rerouting the control flow to another newly created process instance. However, manually inserting such activities is a tedious task and renders the workflow definition unmaintainable as every possible position in the workflow definition has to be supplied with an additional activity. Thus, the WS-BPEL transformer automates the augmentation and the process modeler can stick to his or her original workflow definition.

Figure 10 provides a small example from the software engineering domain (an implement-test-release process). The original workflow definition X is augmented with additional activities DAI1 to DAI4 yielding a workflow definition Xa. This workflow definition can be executed at run time within a workflow instance Xa.1. The process model editor hides the additional activities from the process participant. Assume that during the execution of Implement the need for an additional quality assurance step is identified. The process participant then just embeds an according activity Review within the existing control flow structure. This addition is reflected by the dynamics component inasmuch the DAI2 activity is bound to a newly created instance Ya.1. Each DAI

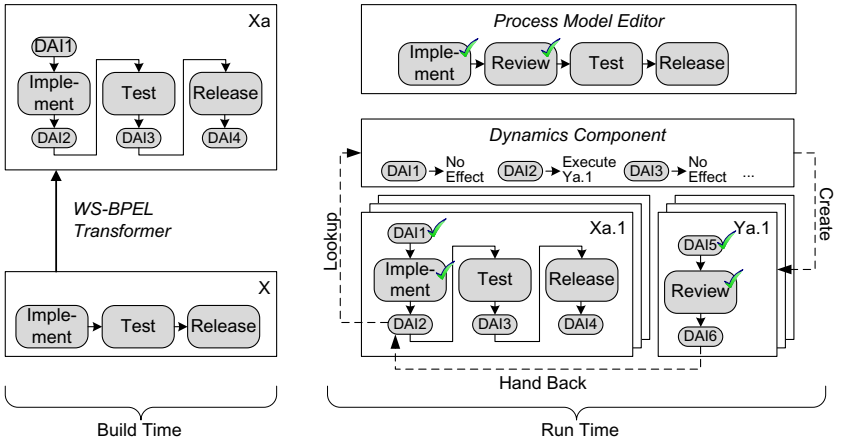


Fig. 10. Realization of Dynamic Adding

activity retrieves binding information from the dynamics component when executed. In the case of DAI2, the new instance Ya.1 is executed such that the Review activity is performed. After completion of Ya.1, the control flow returns to the caller DAI2, and Xa.1 is executed further.

Effectively, the DAI activities together with the binding information of the dynamics component, which is indirectly manipulated by the process participant, simulate an actual dynamic change, which is possible due to late binding.

Dynamic Removal. Complementary to dynamic addition of activities, process participants frequently need to remove mandatory activities. *Dynamic removal* constitutes another dynamics pattern which is supported by the dynamics layer. Yet the overall approach remains the same: At build time, the WS-BPEL transformer augments the workflow definition by additional control flow structures which are used at run time in order to simulate the dynamic removal of activities from a running workflow instance.

As Figure 11 depicts, each original activity is surrounded by a DRD decision activity (diamond). Depending on a dedicated value, which is stored in the dynamics component and evaluated when the respective DRD decision is reached by the control flow at run time, the original activity is either executed (default case) or bypassed. The latter takes place if the process participant has dynamically removed the activity. Again, the generated DRD activities are invisible in the process model editor. Here, dynamic removals are rendered by actually removing the respective activity from the workflow definition.

Dynamic Iteration. Besides the decision elements labeled with DRD, Figure 11 also depicts a decision labeled with DID. This element is generated by the WS-BPEL transformer in order to allow for *dynamic iterations*. These are the equivalents to the feedback flows in AHEAD, i.e. they allow to step back in the control flow and to re-execute activities, which might have been erroneously executed before. At run time, the DID decision works in combination with DRD decisions: Consider the case, where the

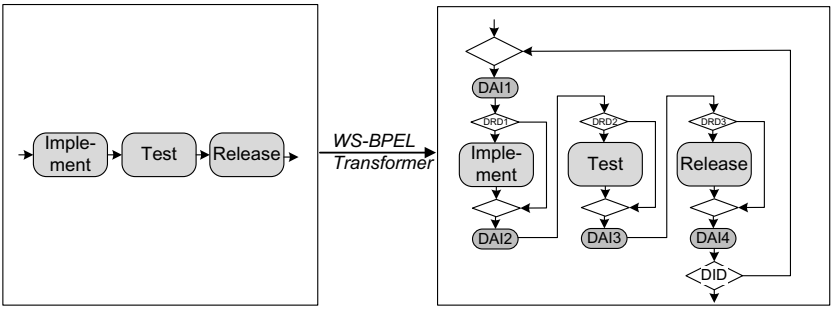


Fig. 11. Transformation for Dynamic Removal and Iteration

workflow instance already has proceeded to Release and the process participant wants to step back to Implement (e.g., because an automated regression test which was not part of the unit Test activity fails in the Release step). Then the value backing the DID in the dynamics component is reset such that the control flow returns to the very beginning of the workflow. In this way, another iteration is started until eventually the final Release step is passed successfully.

Expressive Power of Pattern-Oriented Changes. Interleaved planning and execution calls for maximal freedom concerning the editing of process models (as provided by dynamic task nets in the AHEAD system). In contrast, the dynamics patterns – each constituting a certain kind of deviation – are more restrictive. Regarding the example of Figure 10, it is, e.g., not possible to add the activity Review in parallel to Test (allowing to delegate review and test to different developers working in parallel). Yet, this lack of flexibility does not pose a real problem: The decline of process performance due to the serial instead of parallel execution is neglectable as the dynamic change just affects a single workflow instance.

4.4 Related Work

Many research groups have investigated the development of flexible workflow management systems. Weber et al. [47] provide an overview over the different aspects of flexibility in workflow management systems and define according change patterns. Within the ADEPT project, Reichert et al. [23] investigate dynamic changes to running workflow instances. For this purpose, they use a graph based calculus with some optimizations to ensure the correctness of control flow graphs. The WASA approach [48] is similar to the ADEPT approach. Dynamic changes in workflows are supported by special planning activities which define possible extension points for the workflow. Casati et al. [49] advocate the automated handling of unforeseen process exceptions using a so-called exception-specification language. This language provides additional declarative set-oriented conditions which complement common imperative workflow definitions. All prototypes have in common that they have been developed from the start to support dynamic workflows, i.e. flexibility is integrated a priori. Furthermore, the prototypes use their own modeling languages for workflow definitions instead of standardized languages.

5 Discussion

In this section, we summarize several lessons learned, i.e., conclusions which we have drawn from our work on process support for dynamic development processes. These conclusions are condensed into a set of theses, each of which is discussed briefly.

5.1 Domain-Specific vs. Domain-Independent Management System

Thesis 1 (Domain-Independent Management Systems). *The required functionality of process support tools depends on the actual application domain only to a limited extent. Other process properties such as the level of granularity and the degree of structuring are much more relevant.*

In this respect, we completely agree with [50], Thesis 3. Process support systems cannot be reasonably classified with respect to their application domains. For example, the AHEAD system was applied to multiple engineering disciplines such as mechanical, chemical, and software engineering. Rather, each process support system builds upon certain assumptions concerning the target processes to be addressed. A process support system may be applied to processes satisfying these assumptions, regardless of the specific domain. For example, a workflow management system which has been designed for supporting static business processes may be applied to software processes as long as these processes match the properties assumed by the workflow management system (e.g., structured change request processes, where a well-defined workflow exists which has to be enforced). Furthermore, a process management system like AHEAD may be applied not only to development processes, but also to other processes provided that there is a need for project planning and integrated management of products, activities and resources (e.g., construction of buildings).

Thesis 2 (Ubiquitous Need for Dynamic Changes). *The need for changing processes at run time arises in virtually any application domain. In particular, dynamic changes need to be supported even for structured routine processes.*

In our work, we have primarily studied development processes in different engineering disciplines. However, a small fraction of our work has been dedicated to business processes, as well. In all of these application domains, a strong need for dynamic changes has been identified. The specific requirements for supporting dynamic changes may vary from application to application. However, the mechanisms listed in Subsection 2.5 — ad hoc processes, exception handling, flexible control flows, late binding, interleaving of planning and enactment, or deviations from process model definitions — may be employed in any domain. In particular, we argue that workflow management systems need to support dynamic changes instead of merely enacting static workflows as defined.

5.2 Project vs. Workflow Management

Thesis 3 (Different Focus: Project Planning vs. Process Automation). *Conventional project and workflow management systems focus on different process support functions.*

While project management systems address project planning on a fairly coarse-grained level, workflow management systems automate routine processes typically at a more fine-grained level.

Here, we disagree with Thesis 1 of [50], which basically claims that all kinds of process support tools virtually address the same kinds of problems. In fact, conventional project management systems do not support process enactment, while conventional workflow management systems do not address project planning. Furthermore, the ways processes are modeled in these classes of systems differ considerably. In the case of project management, a plan is built manually at run time. The plan contains only the tasks which actually are to be enacted, and it may be modified at any time during the course of the project. In contrast, a workflow management system instantiates a pre-defined workflow which has to cover all potential execution paths. Furthermore, conventional workflow management systems still fall short of supporting dynamic changes. As a consequence, a project management system cannot replace a workflow management system and vice versa.

Thesis 4 (Integration of Workflow Process Fragments). *Only fragments of the overall development process may be defined as workflows. Thus, these fragments may have to be glued together at the project management level. For supporting integrated process management, project and workflow management have to be integrated or even unified.*

This observation motivated the research presented in Section 3. Development processes may be defined as workflows at best in a fragmentary way. However, if such fragments exist, a workflow management system may provide (partial) process automation. In this case, using a workflow management system improves process support. Those fragments which are relevant at the managerial level have to be embedded into the overall development process (consider, e.g., well-defined and rigorously controlled change request processes in software maintenance). This requires at least an integration of the respective process support systems (as shown in this paper) — or even a unification into a novel process management system, which, however, is not available to date.

5.3 A Priori vs. A Posteriori Integration

Thesis 5 (Economic Value and Necessity). *A posteriori integration allows to retain investments into existing systems. It is an economic necessity in the case of big investments whose replacement would imply a huge development effort. In particular, this applies to management systems for development processes.*

A priori integration means that components of an overall system are designed from the very beginning with a common concept of integration in mind. This approach was realized successfully e.g. in the IPSEN project [51], and it was also applied in the AHEAD system for integrating the management of products, activities, and resources. A priori integration is attractive for several reasons (tight integration, homogeneous development environment, etc.). However, *a posteriori integration* was required by our industrial partners in various projects, and it acted as a driving theme in the IMPROVE project. Companies which have invested into systems for workflow, project, document,

or engineering/product data management need to reuse these systems. Re-development from scratch is not economically feasible.

Thesis 6 (Technical Achievements and Limitations). *A posteriori integration is constrained by limitations imposed by the reuse of existing systems which have been developed independently. Even in the face of these limitations, tight integration may be achieved.*

A posteriori integration is well known to be a challenging task. The functionality of the coupling which may be achieved with the help of a posteriori integration strongly depends on the interfaces provided by the systems to be integrated. However, in the work presented in this paper we succeeded in providing fairly tight integration: Workflows may be monitored in development processes, and dynamic changes may be realized with a conventional workflow management system. This work demonstrates what can be achieved under the constraints of a posteriori integration.

5.4 Barriers of Technology Adoption

As a matter of fact, support technology for development processes has made its way into industrial practice only to a limited extent. In this subsection, we discuss barriers of technology adoption since they are relevant for future work in this area. However, we should stress that these barriers are by no means specific for our own work and have been observed also e.g. in [50, 52].

Thesis 7 (Ease of Use). *Ease of use is an essential requirement to process support systems. Users of process support systems expect light-weight, yet powerful support for dynamic development processes. Unfortunately, the tools which have been developed so far do not meet this requirement.*

Potential users of management systems for dynamic development processes expect that they are supported in their processes by tools which are easy to understand, require only little effort in their use, and yet provide a significant added value. To achieve this, systems are needed which hide the underlying and inherent complexity in managing development processes under a simple and carefully designed user interface. This may be explained by an example from software configuration management: The success of version control systems such as CVS [53] or Subversion [54] is due to the fact that they may be operated through a small set of commands which are simple to use (e.g., checkout, update, and commit). Most process management systems are not as easy to use (this also applies to our own work concerning the AHEAD system).

Thesis 8 (Modeling Effort and Return of Investment). *Modeling of development processes is inherently difficult, in particular if it comes to modeling processes to be enacted. The effort of modeling needs to be balanced against the improved process support achieved with the help of the model.*

In the scenarios which we studied in engineering disciplines, it was always hard to come up with a process definition. Usually, process knowledge was only implicit (i.e., domain

experts had acquired process knowledge which was not documented in any way), and we were faced with the task of defining process definitions which are reusable for a sufficiently large set of development processes. It was inherently difficult to define types of tasks, inputs, outputs, control, data, and feedback flows, etc. For a process definition, both the structure and the behavior need to be specified precisely so that the definition may be used to drive enactment. This is far more challenging than creating a descriptive and informal process definition for documentation or guidance. For gaining acceptance of the prospective users of a process management system, it is crucial that the modeling effort may be balanced against the improved process support such that an appropriate return of investment is achieved. An initial step towards this goal is provided by the AHEAD system, which may be used "out of the box" without any prior modeling at all. However, more research is still required with respect to ease of use and return of investment.

6 Conclusion

We presented a management system for dynamic development processes which integrates the management of products, activities, and resources and provides for seamless interleaving of planning and enactment. Furthermore, we investigated the application of workflow management systems to dynamic development processes and described two complementary solutions. The first solution combines process management and workflow management by integrating workflows into a coherent overall development process. The second solution improves the capabilities of a classical workflow management system by adding components for dynamic changes.

Resuming the topics of the discussion in Section 5, we envision the need of future research activities in the following areas:

Dynamic Processes in Different Application Domains. As we have argued above, process support systems cannot be reasonably classified with respect to their application domains. Rather, each process support system is built for some class of processes which are characterized by structural properties (e.g., degree of structuring, degree of automation, granularity of process support, organizational scale; see [16], Chapter 4). Further work is required to elaborate on these properties and the implications on tool support.

Integrating Project and Workflow Management. From our work, we conclude that project and workflow management systems typically operate at different levels of granularity and satisfy different requirements, e.g., with respect to the dynamics of work processes. While we have developed several integration approaches, we still believe that further work has to be performed to clarify the borderlines between and the cooperation of project and workflow management systems and to investigate the potentials for unification.

A Posteriori Integration. As a matter of fact, there is no way around a posteriori integration, which aims at integrating existing tools into an overall environment providing an added value to its users. We are convinced that integration or extension of existing systems is far superior from an economic point of view and more feasible than building a process management system from scratch, which would have

to cover the whole process scope and would have to compete with existing systems in terms of functionality, maturity, and compliance to standards. While we have successfully realized different models of integration, our work still has to be generalized beyond the integration or extension of specific systems.

Ease of Use and Return of Investment. While a rich variety of mechanisms for supporting dynamic development processes has been proposed and realized, the impact of research on industrial practice is still limited. We have identified ease of use as a critical factor determining technology adoption. More research has to be carried out to provide light-weight, yet powerful support for dynamic development processes which provides a significant return of investment on behalf of the users of process support systems.

Acknowledgments. All work reported in this paper has been performed at Department of Computer Science 3, RWTH Aachen University, under the supervision of Manfred Nagl, who served as a speaker of three consecutive research projects funded by the Deutsche Forschungsgemeinschaft (DFG), namely SUKITS, IMPROVE, and the technology transfer project T6. Support from our industrial partners Generali Deutschland Informatik Services GmbH and Comos Industry Solutions is also gratefully acknowledged.

References

1. Nagl, M., Marquardt, W. (eds.): Collaborative and Distributed Chemical Engineering - From Understanding to Substantial Design Process Support. LNCS, vol. 4970. Springer, Heidelberg (2008)
2. Jablonski, S., Bußler, C.: Workflow Management — Modeling Concepts and Architecture. International Thomson Publishing, Bonn (1996)
3. Bauer, T.: Kooperation von Projekt- und Workflow-Management-Systemen. Informatik - Forschung und Entwicklung 19, 74–86 (2004)
4. Maurer, F., Dellen, B., Bendeck, F., Goldmann, S., Holz, H., Kötting, B., Schaaf, M.: Merging Project Planning and Web-Enabled Dynamic Workflow Technologies. IEEE Internet Computing 4(3), 65–74 (2000)
5. Chan, K., Chung, L.: Integrating Process and Project Management for Multi-Site Software Development. Annals of Software Engineering 14, 115–143 (2002)
6. Bussler, C.: Workflow instance scheduling with project management tools. In: Proceedings of the 9th International Workshop on Database and Expert Systems Applications (DEXA 1998), Washington, DC, USA, pp. 753–758. IEEE Computer Society, Los Alamitos (1998)
7. Comos Industry Solutions (April 2009), http://www.comos.com
8. Intergraph: Smartplant enterprise (March 2010), http://www.intergraph.com/global/de/ppm/spe.aspx
9. Heller, M., Jäger, D., Krapp, C.A., Nagl, M., Schleicher, A., Westfechtel, B., Wörzberger, R.: An adaptive and reactive management system for project coordination. In: Nagl, M., Marquardt, W. (eds.) Collaborative and Distributed Chemical Engineering. LNCS, vol. 4970, pp. 300–366. Springer, Heidelberg (2008)
10. Nagl, M., Westfechtel, B., Schneider, R.: Tool support for the management of design processes in chemical engineering. Computers and Chemical Engineering 27, 175–197 (2003)

11. Heller, M., Jäger, D., Schlüter, M., Schneider, R., Westfechtel, B.: A management system for dynamic and interorganizational design processes in chemical engineering. Computers and Chemical Engineering 29, 93–111 (2004)
12. Krapp, C.A.: An Adaptable Environment for the Management of Development Processes. Aachener Beiträge zur Informatik, vol. 22. Augustinus Buchhandlung, Aachen (1998)
13. Jäger, D.: Unterstützung übergreifender Kooperation in komplexen Entwicklungsprozessen. Aachener Beiträge zur Informatik, vol. 34. Augustinus Buchhandlung, Aachen (2003)
14. Schleicher, A.: Management of Development Processes: An Evolutionary Approach. Informatik. Deutscher Universitäts-Verlag, Wiesbaden (2002)
15. Heller, M.: Dezentralisiertes sichtenbasiertes Management übergreifender Entwicklungsprozesse. Informatik. Shaker Verlag, Aachen (2008)
16. Westfechtel, B.: Models and Tools for Managing Development Processes. LNCS, vol. 1646. Springer, Heidelberg (1999)
17. Heimann, P., Joeris, G., Krapp, C.A., Westfechtel, B.: DYNAMITE: Dynamic task nets for software process management. In: Proceedings of the 18th International Conference on Software Engineering (SE 1996), Berlin, Germany, pp. 331–341. IEEE Computer Society Press, Los Alamitos (March 1996)
18. Hai, R., Heller, M., Marquardt, W., Nagl, M., Wörzberger, R.: Workflow support for interorganizational design processes. In: Marquardt, W., Pantelides, C. (eds.) 16th European Symposium on Computer Aided Process Engineering and 9th International Symposium on Process Systems Engineering, Garmisch-Partenkirchen, Germany. Computer-Aided Chemical Engineering, vol. 21, pp. 2027–2032. Elsevier, Amsterdam (2006)
19. Heller, M., Nagl, M., Wörzberger, R., Heer, T.: Dynamic Process Management Based Upon Existing Systems. In: Nagl, M., Marquardt, W. (eds.) Collaborative and Distributed Chemical Engineering. LNCS, vol. 4970, pp. 733–748. Springer, Heidelberg (2008)
20. Heer, T., Briem, C., Wörzberger, R.: Workflows in dynamic development processes. In: Ardagna, D., Mecella, M., Yang, J. (eds.) Business Process Management Workshops, Milano, Italy. Lecture Notes in Business Information Processing, vol. 17, pp. 266–277. Springer, Heidelberg (2008)
21. Voorhoeve, M., van der Aalst, W.M.P.: Ad-hoc workflows: Problems and solutions. In: Wagner, R. (ed.) Proceedings 8th International Workshop on Database and Expert Systems Applications (DEXA 1997), Toulouse, France, pp. 36–40. IEEE Computer Society Press, Los Alamitos (September 1997)
22. Heinl, P., Horn, S., Jablonski, S., Neeb, J., Stein, K., Teschke, M.: A comprehensive approach to flexibility in workflow management systems. In: Georgakopoulos, D., Prinz, W., Wolf, A.L. (eds.) Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC 1999), San Francisco, CA. ACM SIGSOFT Software Engineering Notes, vol. 24-2, pp. 79–88. ACM Press, New York (March 1999)
23. Reichert, M., Dadam, P.: ADEPTflex — Supporting Dynamic Changes of Workflows Without Loosing Control. Journal of Intelligent Information Systems 10(2), 93–129 (1998)
24. Joeris, G., Herzog, O.: Managing evolving workflow specifications. In: Proceedings of the International Conference on Cooperative Information Systems (CoopIS 1998), New York, NY, pp. 310–321. IEEE Computer Society Press, Los Alamitos (1998)
25. Wörzberger, R., Ehses, N., Heer, T.: Adding support for dynamics patterns to static business process management systems. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 84–91. Springer, Heidelberg (2008)
26. Wörzberger, R.: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme. Aachener Informatik-Berichte, Software Engineering, vol. 2. Shaker Verlag, Aachen (2010)

27. Jäger, D., Schleicher, A., Westfechtel, B.: Using UML for software process modeling. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC/FSE 1999. LNCS, vol. 1687, pp. 91–108. Springer, Heidelberg (1999)
28. Schleicher, A., Westfechtel, B.: Beyond stereotyping: Metamodeling approaches for the UML. In: IEEE Hawaii International Conference on System Sciences (HICSS-34), Minitrack Unified Modeling Language: A Critical Review and Suggested Future, Maui, HI, pp. 1–10 (January 2001)
29. Schneider, R., Westfechtel, B.: A scenario demonstrating design support in chemical engineering. In: Nagl, M., Marquardt, W. (eds.) Collaborative and Distributed Chemical Engineering. LNCS, vol. 4970, pp. 39–60. Springer, Heidelberg (2008)
30. Lawrence, P. (ed.): Workflow Handbook. John Wiley & Sons, Chichester (1997)
31. Workflow Management Coalition: Workflow process definition interface – XML process definition language (XPDL), version 1.0 (April 2002), http://www.wfmc.org/standards/XPDL.htm
32. OASIS: OASIS Web Services Business Process Execution Language Version 2.0 (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
33. Finkelstein, A., Kramer, J., Nuseibeh, B. (eds.): Software Process Modelling and Technology. Advanced Software Development Series. Research Studies Press (John Wiley & Sons), Chichester, UK (1994)
34. Derniame, J.C., Baba, A.K., Wastell, D. (eds.): Software Process: Principles, Methodology, and Technology. LNCS, vol. 1500. Springer, Heidelberg (1999)
35. Hagen, C., Alonso, G.: Exception handling in workflow management systems. IEEE Transactions on Software Engineering 26(10), 943–958 (2000)
36. Jablonski, S.: Do we really know how to support processes? Considerations and reconstruction. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65^{th} Birthday. LNCS, vol. 5765, pp. 393–410. Springer, Heidelberg (2010)
37. Bolcer, G.A., Taylor, R.N.: Endeavors: A process system integration infrastructure. In: Proceedings of the 4th International Conference on the Software Process, Brighton, England, pp. 76–89. IEEE Computer Society Press, Los Alamitos (December 1996)
38. Bandinelli, S., Fuggetta, A., Ghezzi, C.: Software process model evolution in the SPADE environment. IEEE Transactions on Software Engineering 19(12), 1128–1144 (1993)
39. Jaccheri, M.L., Conradi, R.: Techniques for process model evolution in EPOS. IEEE Transactions on Software Engineering 19(12), 1145–1156 (1993)
40. Cugola, G., Nitto, E.D., Fuggetta, A., Ghezzi, C.: A framework for formalizing inconsistencies and deviations in human-centered systems. ACM Transactions on Software Engineering and Methodology 5(3), 191–230 (1996)
41. Cugola, G.: Tolerating deviations in process support systems via flexible enactment of process models. IEEE Transactions on Software Engineering 24(11), 982–1001 (1998)
42. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. In: Thalheim, B. (ed.) ER 1996. LNCS, vol. 1157, pp. 438–455. Springer, Heidelberg (1996)
43. Kradolfer, M., Geppert, A.: Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In: Proceedings of the International Conference on Cooperative Information Systems (CoopIS 1999), Edinburgh, pp. 104–114. IEEE Computer Society Press, Los Alamitos (September 1999)
44. Maurer, F., Dellen, B., Bendeck, F., Goldmann, S., Holz, H., Kötting, B., Schaaf, M.: Merging project planning and web-enabled dynamic workflow technologies. IEEE Internet Computing 4(3), 65–74 (2000)
45. Enhydra.org Community: Enhydra Shark – Java Open Source XPDL workflow, version 1.1-2 (2005), http://www.enhydra.org/workflow/shark/index.html

46. Weisemöller, I.: Verteilte Ausführung dynamischer Entwicklungsprozesse in heterogenen Prozessmanagementsystemen. Master's thesis, RWTH Aachen University (2006)
47. Weber, B., Rinderle, S.B., Reichert, M.: Change patterns and change support features in process-aware information systems. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 574–588. Springer, Heidelberg (2007)
48. Vossen, G., Weske, M.: The WASA approach to workflow management for scientific applications. In: Dogac, A., Kalinichenko, L., Ozsu, M.T., Sheth, A. (eds.) Workflow Management Systems and Interoperability. ASI NATO Series, Series F: Computer and Systems Sciences, vol. 164, pp. 145–164. Springer, Berlin (1999)
49. Casati, F., Ceri, S., Paraboschi, S., Pozzi, G.: Specification and implementation of exceptions in workflow management systems. ACM Transactions on Database Systems 24(3), 405–451 (1999)
50. Conradi, R., Fuggetta, A., Jaccheri, M.L.: Six theses on software process research. In: Gruhn, V. (ed.) EWSPT 1998. LNCS, vol. 1487, pp. 100–104. Springer, Heidelberg (1998)
51. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996)
52. Cugola, G., Ghezzi, C.: Software processes: a retrospective and a path to the future. Software Process: Improvement and Practice 4(3), 101–123 (1998)
53. Vesperman, J.: Essential CVS. O'Reilly, Sebastopol (2006)
54. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly, Sebastopol (2004)

An Extensible Modeling Language for the Representation of Work Processes in the Chemical and Process Industries

Ri Hai, Manfred Theißen, and Wolfgang Marquardt

AVT – Process Systems Engineering, RWTH Aachen University,
Templergraben 55, 52056 Aachen, Germany

Abstract. Expressive models of the work processes performed in the chemical and process industries provide a basis for diverse applications like work process documentation, analysis, and enactment. In this contribution, we present a generic modeling language for different types of work processes to allow for their integrated representation in the life cycle of a chemical plant. Further, the generic language allows for extensions specific to certain types of work processes. For two important types – design and operational processes – such extensions have been elaborated. These extensions enable the adequate representation of the context of a work process that strongly depends on the process type: for instance, the specification of a chemical plant is a product of a design process, whereas the plant takes the role of a resource during an operational process. This contribution also briefly introduces a modeling tool developed by our group for applying the modeling language in industrial practice.

Keywords: work process, design process, operational process, process modeling language, ontology.

1 Introduction

A *work process* is a collection of interrelated actions in response to an event that achieves a specific result for the customer of the work process. This definition was originally proposed by Sharp and McDermott [1] for *business processes*. However, the term *business process* is ambiguous as there are different definitions in different communities such as in *business process engineering* (e.g., [2]) or in *workflow management* (e.g., [3]). Moreover, the term is typically applied exclusively for work processes which are completely determined and therefore can be planned in a detailed manner. Examples include the processing of a credit transfer in a bank or of a claim in an insurance company. In consequence, many work processes in chemical engineering, in particular design processes, are commonly not considered as business processes. To overcome the ambiguities, we prefer the term *work process* as defined above.

In the life cycle of a chemical plant, several types of work processes are performed, including design processes and operational processes. Whereas design

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 655–682, 2010.

processes aim at the specification of chemical products, production processes, or operating procedures for a plant, operational processes target at establishing, maintaining or preventing certain process conditions during production. Typical examples of operational processes are the start-up of a continuously operated plant or the operation of a multi-purpose batch plant. Our research group has investigated work processes in the chemical industry for more than a decade [4].

An important result of this work is the insight that there is a need for substantial support for engineers and technicians with a background in chemical engineering to create simple and informal models of work processes (e.g., operational procedures for a plant), to enrich these models with further details next and to finally increase their level of formality (e.g., in order to automate an operational procedure). We have performed several case studies focusing on the modeling of both design and operational processes [5,6]. The empirical results of these case studies have motivated the specification of a *modeling framework* comprising

- a modeling procedure as a guide for modelers that need a work process model for a certain application such as documentation, analysis, or (partial) automation of a work process [5];
- a modeling language, i.e., a meta-model providing the modeling elements to represent work processes;
- prototypical modeling tools as a prerequisite for the practical application and validation of our research results in an industrial environment.

The *modeling procedure* proposes several iterations with increasing levels of generality, detail, and formalization, as required by the application. The *level of generality* refers to the number of different work processes covered by a model, whereas the *level of detail* refers to the amount of information captured. Finally, the *level of formalization* refers to the representation of this information (e.g., by means of textual annotations or by using some model elements with well-defined semantics). A more elaborated description of the levels of generality, detail, and formalization has been published elsewhere [8].

The representation of work processes requires an adequate *modeling language*, which is the focus of this contribution. In Sec. 2, we discuss the requirements for such a modeling language. An overview of existing modeling languages as well as their evaluation with respect to industrial requirements is given in Sec. 3. We will see that existing languages partially meet the requirements, but none of them, to our knowledge, satisfactorily fulfills the requirements in a comprehensive manner. The following two sections introduce the *Work Process Modeling Language* (WPML), a language developed by our group based on existing approaches such as the C3 language [5,7,8] and the *activity diagram* of the *Unified Modeling Language* (UML, [9]). Sec. 4 deals with the behavioral aspect of work processes and the modeling elements offered by WPML that go beyond existing languages; these modeling elements are independent of the type of the work process under consideration. Sec. 5 is about WPML's support for the functional aspect of work processes, which is neglected by most existing languages. The function of a work

process, and consequently the modeling elements required for representing functional aspects, strongly depend on the process type. For two important types of work processes in chemical engineering, namely design processes and operational processes, functional modeling elements have been elaborated. An overview on a prototypical *modeling tool* is given in Sec. 6.

2 Requirements for a Modeling Language

An adequate modeling language for work processes must fulfill several requirements which result not only from the characteristics of the work processes under consideration, but also from the modeling procedure sketched above. These requirements are discussed in the following.

2.1 Expressiveness

The expressiveness of a modeling language refers to the different aspects of or views on an object—a work process in our case—that can be represented by the language. The focus of this contribution is on behavioral and functional aspects of a work process. Further aspects, such as the actors performing a work process, the technical resources required for their execution (e.g., see [8]), or the complex decisions typically involved in engineering design processes [10,11], are not discussed in this paper.

Behavioral aspects. The focus of most work process models is the representation of process behavior, i.e., the dependencies between the elementary steps—called *actions* in the following—with respect to causal conditions (e.g., *if action A is executed, then action B must also be executed*) and temporal restrictions (e.g., *B is executed after A is finished*). Existing modeling languages often force the modeler to create a *complete* behavior specification from the beginning of the modeling process.

As an example, we consider Aspen Batch Process Developer [12], an engineering software tool for the *simulation* of operational processes in a batch plant. In industrial practice, this simulation tool is also often used as a *modeling* tool to create a first draft of a batch process. At this stage, many details of the batch process are still undefined. For instance, a chemical engineer may want to provide the information that a reactor should be charged with two reactants A and B in a yet undefined order. When creating a model in the tool, the engineer is *forced* to specify a certain order because the underlying modeling language does not allow to omit that information. In consequence, the work process model is likely to be enriched with uncertain or even accidental content, while other feasible alternatives are excluded.

During the design of an operational process, incomplete behavior specifications are useful intermediate steps on the way towards a complete specification, which is required to realize process automation system and ultimately to operate a chemical plant. In case of work process models in engineering design, complete

behavior specifications are infeasible in general because design is an inherently creative process, which cannot be predetermined in principle.

In conclusion, a modeling language has to allow the representation of work process models with an incomplete behavior specification. In particular, there is a need for two kinds of abstraction [8]: *Structural abstraction* refers to the freedom to omit information about the conditions under which an action is performed, and *temporal abstraction* refers to the freedom to omit information about the temporal relations between actions *if* they are executed (e.g., by permitting a partial overlap of two subsequent actions).

Functional aspects. According to the definition of the term *work process* in the introduction, the *function* of a work process is to create a specific result. However, most modeling languages focus on the behavioral aspect of a work process and pay little attention to the representation of the function of an entire work process or of the actions within it. Usually, the functional aspect is exclusively specified by means of auxiliary constructs such as textual annotations.

Textual descriptions of functional aspects are advantageous in the early stages of a modeling process because a modeler can phrase a text more easily than choose a specific modeling concept and insert it into a model. There are also model applications in which a functional representation beyond textual descriptions does not provide any additional benefit. An example is the model of a simple operational process for a lab-scale plant, which only serves as a reference for the operator to execute the process.

However, explicit representations of functional aspects enable more advanced applications of work process models, including automatic model checks and model transformations for target applications that strongly depend on functional aspects. Automatic model checks rely on the level of formalization as discussed below. As for model transformations, we consider the example of an operational process that includes the heating of some process material. If the *heating* function is represented by explicit modeling elements rather than by ambiguous textual annotations, a transformation of the model into a simulation model for Aspen Batch Process Developer is possible.[1]

The function of a work process (or of an action) is typically associated with certain objects as prerequisites or outcomes; consequently, a complete modeling of the functional aspect should include these objects (see also [13,14]). To give an example, the specification of a *heating* function for an action is rather incomplete without the specification of the process material to be heated.

2.2 Formalization

A model is *formal* if it's semantics is defined by a mathematical formalism. *Formalization* refers to the transformation of an informal model into a formal one. A trivial prerequisite for a formal work process model is a formal modeling

[1] The tool does not support an abstract 'general' process step, but only concrete types such as *heating* and *cooling*.

language, where the semantics of the modeling elements are defined formally. At the same time, a modeling language should not compel the user to a certain level of formalization. Rather, a set of modeling elements should be provided at a level of formalization that is imposed by the intended application of the model. In particular, it should be possible to represent some aspects formally, but other aspects informally, e.g., by means of simple textual annotations.

Formal representations facilitate advanced applications of work process models like automatic model checks or the derivation of implicit knowledge from explicitly defined knowledge. As an example, we reconsider the modeling of a *heating* function, which could be defined as a modeling element for a *temperature change* resulting in an end temperature higher than the start temperature. Based on a formal representation of this definition, the automatic detection of a semantic error in a model with a *heating* step from 80°C to 60°C is possible. In a similar way, it could be derived that a *temperature change* step from 60°C to 80°C is actually a *heating* step; such derived knowledge would then be available for applications like model transformations.

3 Modeling Languages for Work Processes

There is a plethora of modeling languages for work processes, and a complete review of the existing approaches is beyond the scope of this contribution. Instead, some representative languages are discussed and evaluated with respect to the requirements above. First, *generic* languages are addressed which have been developed for any type of work process (or at least a wide range of types), followed by *specific* languages for operational and design processes.

3.1 Generic Modeling Languages

The *activity diagram* of UML [9] is probably the best-known graphical modeling language for work processes. The language specification comprises typical modeling elements like actions and control flows as well as concepts to represent parallel or alternative branchings of the control flow (i.e., split and decision nodes).[2] The behavioral semantics of UML activities is defined by an informal textual description of the *token flow* in an activity diagram. Hence, the activity diagram can be interpreted as a kind of Petri net, but is not formally defined as such.

[2] In addition to the general action element, more specific subclasses are defined in UML such as *CreateObjectAction* or *WriteVariableAction*. Given that UML is rooted in software engineering, these specific action types are not suitable for the representation of work processes in chemical engineering. However, usage of the special action types in UML is not mandatory, and in practice it is restricted to rather advanced applications like *Model Driven Architecture* [15], which aims at the automated generation of code based on UML models. This justifies the classification of UML activity diagrams as a generic modeling language.

Petri nets are a widely-used family of graphical modeling languages [16,17]. Their strengths include the inherent support of concurrency and resource allocation concepts and the formal mathematical foundation, which eliminates any ambiguity and allows the application of Petri nets for the automated analysis and execution of work processes. The direct application of Petri nets to model complex work processes is often not practical, but they have been applied to specify the semantics of other modeling languages. Störrle and Hausmann [18], for instance, propose formal behavioral semantics for (part of) UML activities by means of a well-defined translation to Petri nets. Furthermore, the semantics of *Yet Another Workflow Language* (YAWL, [19]) is based on a direct mapping of its modeling elements to Petri nets.

The *Process Specification Language* (PSL, [20]) is an ontology-based language for the automatic exchange of information about work processes between different applications and to enhance their interoperability. The semantics of PSL is defined explicitly in first order logics. An outline of a formal specification of UML activities by means of PSL has been published [21] to illustrate its formalization capabilities.

The graphical representation and semantics of the *Business Process Modeling Notation* (BPMN, [22]) are similar to that of UML activity diagrams. Even though its name suggests a focus on business processes, the language is rather generic and allows to represent different types of work processes. In particular, BPMN addresses the creation of process models for workflow execution and automation; for this purpose mappings from BPMN to executable languages like the *Business Process Execution Language for Web Services* [23] or *XML Process Definition Language* [24] have been defined [22,25].

IDEF0 of the *IDEF* family (*Integrated Definition Methods*, [26]) is also widely used for work process modeling, especially in the manufacturing domain. IDEF0 contains a set of generic modeling concepts and allows a "*structured representation of the functions, activities or processes within the modeled system or subject area*" [27]. Temporal relations between actions are not included in IDEF0.

Batres and Naka [28] propose a *process plant model* which can be used to represent the structure of a plant and the processed material, including its physical behavior. Further, the process plant model contains a *management and operation ontology* which provides concepts such as *plan*, *activity*, *action* or *activity-performer* for describing work processes. Although the process plant model is mainly used for representing operational processes, it can be used to represent different kinds of work processes. The process plant model is implemented in a development environment called Ontolingua [29], which enables to define the semantics of modeling concepts explicitly and formally.

3.2 Specific Modeling Languages for Operational Processes

Currently, diverse modeling languages are used to represent operational processes (cf. [30]). These languages can be divided into two groups. The first group contains generic modeling languages such as Petri nets or UML. The focus of

this subsection is on the second group, which includes languages developed exclusively or primarily for operational processes.

The IEC 61131-3 standard [31] defines several textual and graphical languages for the automation of operational processes. These languages are considered primarily as programming languages rather than modeling languages.

The VDI/VDE 3682 guideline [32] seeks to facilitate the formal representation of a production process throughout its life cycle. To keep the process representations simple and understandable for engineers from different disciplines as well as for plant operators, the guideline covers only a small set of modeling elements. Further, this guideline also supports the representation of the objects involved in a *process* and their relations by means of UML.

ANSI/ISA-S88 [33] addresses batch process control. The standard defines a layered structure for physical models of plants, process models, and procedural control models. Four types of recipes are defined, including general, site, master, and control recipes. Although the standard itself does not define a modeling language, it provides a useful frame for representing batch process operations.

Beyond these well established standards, current and recent research efforts also deal with the representation of operational processes. For example, on the basis of the ANSI/ISA-S88 standard, Viswanathan *et al.* [34] propose an approach for the synthesis of control recipes. The approach enables to capture the behavioral aspect of an operational process by using Grafchart, a variant of Petri nets. Moreover, domain specific knowledge about chemical plants, chemical processes, and operational processes is included.

Gabbar *et al.* [35] describe a *recipe formal definition language* (RFDL) to support the development of operating procedures of chemical batch plants. RFDL statements are composed of classified keywords, which are linked to domain knowledge, including knowledge about chemical plants, process materials, and operational procedures.

3.3 Specific Modeling Languages for Design Processes

The representation of design processes has been an active research area in the past. Some approaches aim at a framework for analyzing and understanding design processes (e.g., [36,37]). Other approaches focus on detailed models of design artifacts to facilitate design processes, but do not directly address the design process itself [38,39]. Further, there are some guidelines for design processes in the domain of chemical engineering (e.g., [40,41]). In the following, some representative modeling languages for design processes are introduced briefly.

Gorti *et al.* [42] have developed a generic model for representing domain independent design processes. On the one hand, artifacts produced during a design process can be represented under functional, structural, and behavioral aspects. On the other hand, *design processes* can be represented as *process objects* containing process-related components like *goal*, *plan*, or *context*.

The *Knowledge Based Design System* (KBDS, [43]) has been developed to support the design of chemical plants. Based on an exploration-based model of design and the hierarchical decision procedure of Douglas [41], design processes

are represented as networks of design objectives, alternatives, and models. Design alternatives of chemical plants can be represented on different levels of detail. Further, design alternatives are related to both the design objectives and simulation models.

The C3 language (*cooperation*, *coordination*, and *communication*) is a simple graphical modeling language with a special focus on the requirements imposed by weakly-structured design processes [5,7,8]. In the interest of high usability, C3 provides a rather restricted number of modeling elements. However, the user is free to extend the language if needed. In addition to conventional modeling concepts for predetermined processes, C3 provides concepts for temporal and structural abstraction.

CLiP [13,44] is an information model providing a conceptualization of the domain of chemical process design. *Process Models* is a partial model of CLiP covering different types of work processes, such as the development of mathematical models or the design of chemical processes [45]. The model comprises a *meta class layer* which provides the concepts required for generalized processes, and a *simple class layer*, which enables the representation of more concrete design processes. Design processes are modeled as iterations of *synthesis*, *analysis*, and *decision* activities, linked by auxiliary activities [46].

3.4 Evaluation

Several of the modeling languages introduced above enable the representation of process behavior, but only a small subset provides a formal foundation (Petri nets, PSL, the languages of IEC 61131-3, and the approach by Viswanathan *et al.*). Only C3 supports temporal and structural abstraction, but lacks a formal definition of these concepts.

The functional aspect of a work process is covered by several approaches [13,34,28,35,43,44]. Interestingly, most of them also provide a detailed model of the objects involved in a work process. The approach proposed by Batres and Naka [28] is the only approach providing a formal specification of the functional aspect.

In short, the existing approaches are too restrictive with respect to process behavior, since they do not support behavioral abstraction. On the other hand, most approaches lack an explicit and formal representation of the functional aspect of work processes.

4 Behavioral Semantics of WPML

The Work Process Modeling Language WPML has been designed to address the weaknesses of the existing modeling languages discussed above. The core of the language provides modeling elements to represent the behavioral aspect of work processes. This part is independent from any application domain; similar as UML acitivy diagrams, it contains generic concepts to represent actions, decisions, concurrency, etc., which are required for any type of work process.

As argued above, functional modeling is required for diverse applications. Expressive functional models are domain-specific (e.g., the function to *heat* some material in a chemical plant). Thus, the domain-independent core of the language does not consider functions beyond the trivial aspect that an action in a work process serves to fulfil one or several functions.[3] Instead, WPML can be extended with domain-specific modules that enable functional models with the expressiveness required by an application. This section focuses on the domain-independant behavioral semantics of WPML. Two examplary domain-specific extensions are discussed in Sec. 5.

Both the WPML meta model (i.e., the domain-independent core and optional domain-specific extensions) and WPML instance models (i.e., the work process models created by a user) are represented in the Web Ontology Language OWL [47]. OWL is a language for knowledge representation with a formal foundation in the domain of description logics [48]. So-called *reasoners* (e.g., Pellet [49]) can be used to exploit the formal definition of the language for applications like consistency checks for work process models or to derive implicit knowledge (which, for instance, substantially reduces the effort to implement the transformation of models into other formats [50]). In the following, we will use a notation similar to that of UML class diagrams to represent parts of WPML although there are important differences between the semantics of OWL and that of UML.

4.1 Basic Elements of WPML

One goal of WPML is to give a formal definition of several behavior-related concepts of the C3 notation for cooperative work processes, which have proven to be valuable in several academic and industrial case studies [5,6], but that suffer from their ambiguous semantics. An informal description of WPML has been published elsewhere ([8], called *Process Ontology* there), including a discussion of its capabilities with respect to temporal and structural abstraction. The following discussion focuses on the formal specification of process behavior. It is restricted to a simplified subset of WPML, which nevertheless demonstrates the basic idea of the chosen approach. A more detailed introduction to WPML can be found in [50]. In addition, a complete specification of the language is in preparation; it will also cover the *roles* by which actions are executed, the *objects* produced or consumed in actions, and the *object flows* that link actions with objects.

In Fig. 1 the basic structure of a WorkProcessModel in WPML is shown. It is a graph that isComposedOf WorkProcessElements, i.e., WorkProcessNodes and WorkProcessArcs. Each arc is from exactly 1 WorkProcessNode to exactly 1 WorkProcessNode[4]. The only subclass of WorkProcessArc discussed here is the ControlFlow. Table 1 lists several concrete subclasses of WorkProcessNode and

[3] To a certain extent, functions could be represented on a domain-independent, but very abstract level (e.g., production, transformation, or consumption of material or information).

[4] The notation used in Fig. 1 and Table 1 (e.g., from exactly 1) is the Manchester OWL Syntax proposed in [51] as a human-readable alternative for OWL class expressions.

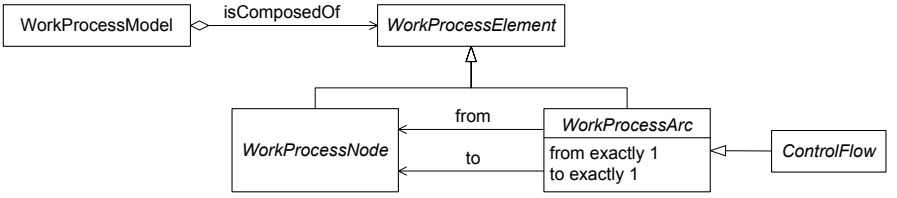


Fig. 1. The basic elements of a WorkProcessModel in WPML

ControlFlow. For the nodes, restrictions for their connection by ControlFlows are given; for instance, an Action must be the origin of exactly one ControlFlow (it must be the from node of exactly 1 ControlFlow), and a final node must not be the origin of a ControlFlow.

4.2 An Introductory Example

Figure 2 shows an example of a simple WorkProcessModel in a graphical notation similar to that of UML activities (the symbols for nodes and arcs are given in Table 1). The model describes the development process for chemical processes in an R&D department on a very coarse-grained level. The first Action a1 is to create a block flow diagram (*BFD*), a schematic representation of a chemical process. In a BFD, blocks or rectangles represent the unit operations or groups of unit operations in the chemical process. Next, more detailed process flow diagrams (*PFD*) must be made for the different parts of the BFD. This can either be done by a contractor or in the same R&D department. For the former case, only a single Action a2 (*Outsource PFD creation*) is given; further details on how the PFDs are created are left to the contractor. For the latter case, two Actions explicitly require that a PFD for the *reaction unit* (a3) and another PFD for the *separation unit* (a4) are created.

In WPML, like in other flow diagrams such as UML activity diagrams, a ControlFlow between two Actions expresses a causal dependency (*if* the first Action is executed, then also the second one must be executed). Conventional languages only include a single control flow, which, in addition to the causal dependency, expresses a temporal dependency (*when* the first Action is finished, the second starts[5]). This is often an unacceptable restriction for models of design or development processes. To this end, there are several types of ControlFlows in WPML, each resulting in different temporal relations between the Actions connected by a flow. In Fig. 2, two different types of control flows are used. The StrictControlFlow is the WPML equivalent of conventional control flows, which require the previous action to be completed before the subsequent action starts; it is drawn as a simple directed arrow. The OverlappingControlFlow allows

[5] It is often ill-defined whether the second Action must start immediately or whether there may be a temporal gap between the Actions.

Table 1. Concrete subclasses of WorkProcessNode (first section) and WorkProcessArc (second section)

Class	Symbol	Restrictions
Action		inv(from) exactly 1 ControlFlow and inv(to) exactly 1 ControlFlow
InitialNode		inv(from) exactly 1 ControlFlow and inv(to) exactly 0 ControlFlow
FinalNode		inv(from) exactly 0 ControlFlow and inv(to) exactly 1 ControlFlow
DecisionNode		inv(from) min 2 ControlFlow and inv(to) exactly 1 ControlFlow
MergeNode		inv(from) exactly 1 ControlFlow and inv(to) min 2 ControlFlow
ForkNode		inv(from) min 2 ControlFlow and inv(to) exactly 1 ControlFlow
JoinNode		inv(from) exactly 1 ControlFlow and and inv(to) min 2 ControlFlow
StrictControlFlow		
OverlappingControlFlow		

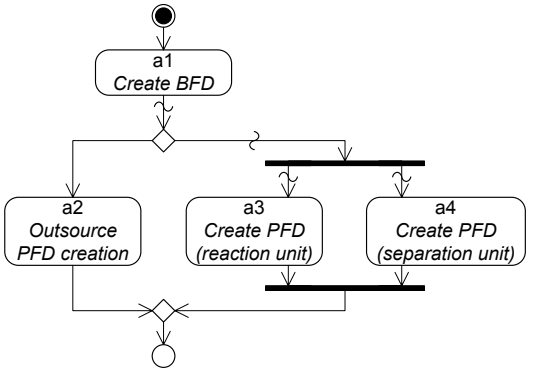


Fig. 2. A simple WorkProcessModel

a subsequent Action to begin while the preceding Action is still executed; it is drawn as a directed arrow decorated with a tilde symbol (˜).

This way, the semantics of WPML separates causality from temporal relations. Causality refers here to the necessary and sufficient conditions for executing an Action. *Causality* is treated similarly in WPML and in UML activity diagrams.

The causal relations between the Actions in the example model are equivalent to those in the UML activity diagram we would get if we replaced the special WPML control flows by the standard UML control flow: *If* the entire work process is executed, then a1 is executed. *If* a1 is executed, then either a2 or both a3 and a4 are executed.

The admissible *temporal relations* are indicated by the different types of control flows in WPML. Above, we have stated that a ControlFlow restricts the temporal relations between Actions connected by the flow. Note that the OverlappingControlFlows in the example do not directly connect Actions, but also control nodes such as the DecisionNode or the ForkNode. However, each OverlappingControlFlow is part of one or several *control flow paths* between Actions. For instance, the OverlappingControlFlow between a1 and the DecisionNode is the start of a path of two control flows from a1 via the DecisionNode to a2; the second control flow is a StrictControlFlow. The temporal restriction between a1 and a2 is defined as the more restrictive one of the two control flows in the path, i.e., the StrictControlFlow. Hence, the model contains the statement that *if* a1 and a2 are executed, i.e., if the PFDs are created by a contractor, then there must be no temporal overlap with the creation of the BFD. Given the obstacles of concurrent engineering across organizational boundaries, this is a reasonable restriction.

We get similar statements for the other control flow paths starting at a1: *If* a1 and a3 are executed, i.e., if the PFD for the reaction unit is created in-house, then there may be a temporal overlap—because the most restrictive control flow is an OverlappingControlFlow. Likewise, *if* a1 and a4 are executed, then there may also be a temporal overlap.

This example demonstrates the capibility of WPML for *temporal* abstraction: Whereas *causal* relations between the Actions are represented similar as in other languages, the *temporal* relations offered by WPML go farly beyond these languages. As for *structural* abstraction, WPML provides a *blob* element, which basically specifies a set of Actions and includes constraints for their occurrences (e.g, *at least 2 and at most 3 of the* Actions *must occur*). For further details on *blobs*, see [7,8].

4.3 Basic Principles of the Formal Specification of WPML

The complete formal specification of the behavioral semantics of WPML is beyond the scope of this contribution. Instead, we describe the basic principles of the specification, which are based on a combination of Petri nets and axioms in first-order logics. As the semantics of the different variants of Petri nets are typically defined in a formal way, it should be possible to create a formal specification of WPML based exclusively on logical axioms[6]. However, this is not the

[6] A WPML semantics based solely on first-order logics would be similar to the approach of the Process Specification Language [20]. It may even be possible to define WPML using PSL, but several concepts of WPML—in particular the idea of overlapping control flows—have no equivalent in PSL and, in our opinion, this task would require substantial extensions or even modifications of several parts of PSL.

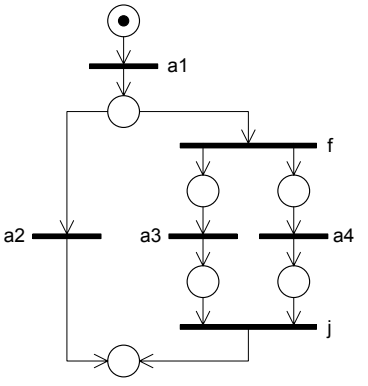


Fig. 3. State-transition net corresponding to the WPML model in Fig. 2

focus of our work; we are mainly interested in an application-oriented way to give a precise definition of WPML.

By an approach similar to that of Störrle and Hausmann [18] for UML activity diagrams, a mapping from WPML models to Petri nets is defined. The Petri net dialect chosen is the colored hierarchical Petri net as defined by Jensen [17]. To simplify the presentation in this overview, we use a simple state-transition net here. The state-transition net corresponding to the WPML model in Fig. 2 is shown in Fig. 3: Each Action is transformed in a transition (denoted by a1, ..., a4). The DecisionNode has become a place with several outgoing arcs, the MergeNode node a place with several incoming arcs. Finally, the ForkNode and the JoinNode are transformed in auxiliary transitions (f, j). It can easily be verified that the net represents the *causal* dependencies of the Actions as discussed above: The topmost place, representing the StartNode of the WPML model, contains a single token. The only active transition is a1; the firing of the transition represents an execution or *occurrence* of Action a1. Then both transitions a2 and f are active. The case that a2 fires is interpreted as an occurrence of a2. Alternatively, f can fire, which will eventually lead to the firing of both a3 and a4 (in an undefined order).

The semantics of *temporal relations* is not involved in the Petri net translation and, in particular, the order in which transitions in the Petri net fire must not be interpreted as the order in which Actions occur. Instead, an approach based on axioms in first-order logics is used. These axioms impose restrictions on auxiliary objects called WorkProcessOccurrences, which can be interpreted as traces of the transitions fired in the Petri net. Like WorkProcesses, WorkProcessOccurrences are graphs composed of elements like ActionOccurrences and DecisionOccurrences[7]. An ActionOccurrence has two functional properties start

[7] It is essential to distinguish between the modeling elements in WPML, e.g., Action, and the auxiliary objects, e.g., ActionOccurrence. In more complex work processes than in the example discussed here, there may be several occurrences of the same element in the model, for instance several occurrences of a single Action in a loop.

and end, whose range is TimePoint. The concept of TimePoints has been adopted from PSL [20]; here, the existence of a strict total order before_TP on TimePoints is essential. We require that for any ActionOccurrence, its start must be before_TP its end.

In Fig. 4, several examples of WorkProcessOccurrences are given. wpo1 is composed of a StartOccurrence, two ActionOccurrences ao1 and ao2, an EndOccurrence, and three StrictControlFlowOccurrences. To simplify the presentation, the TimePoints are represented as integers, and we assume that the before_TP order is isomorphic to the *smaller-than* relation of their integer representations (i.e., 2 before_TP 4, ...). Hence, the ActionOccurrences in the figure respect the condition introduced above (start before_TP end).

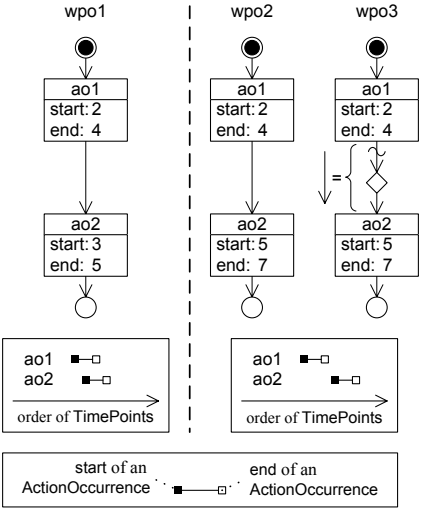


Fig. 4. Graphical depiction of three WorkProcessOccurrences. wpo1 is invalid because neither of the conditions ao1 before ao2 or ao1 meets ao2 holds. wpo2 and wpo3 are valid.

We can easily define thirteen relations between ActionOccurrences in an analogous manner to the time interval relations by Allen [52]. For instance, an ActionOccurrence ao1 meets an ActionOccurrence ao2 iff the end of ao1 and the start of ao2 are equal,

$$\forall ao1, ao2 : \mathsf{ActionOccurrence}(ao1) \wedge \mathsf{ActionOccurrence}(ao2)$$
$$\Rightarrow (\mathsf{meets}(ao1, ao2) \Leftrightarrow \mathsf{end}(ao1) = \mathsf{start}(ao2)) \quad ,$$

and similarly

$$\forall ao1, ao2 : \mathsf{ActionOccurrence}(ao1) \wedge \mathsf{ActionOccurrence}(ao2)$$
$$\Rightarrow (\mathsf{before}(ao1, ao2) \Leftrightarrow \mathsf{before}(\mathsf{end}(ao1), \mathsf{start}(ao2))) \quad ,$$

$$\forall ao1, ao2 : \mathsf{ActionOccurrence}(ao1) \wedge \mathsf{ActionOccurrence}(ao2)$$
$$\Rightarrow (\mathsf{overlaps}(ao1, ao2) \Leftrightarrow \mathsf{before}(\mathsf{start}(ao1), \mathsf{start}(ao2)) \wedge$$
$$\mathsf{before}(\mathsf{start}(ao2), \mathsf{end}(ao1)) \wedge$$
$$\mathsf{before}(\mathsf{end}(ao1), \mathsf{start}(ao2))) \quad .$$

The temporal relations imposed by the different ControlFlows are defined by axioms which restrict the valid time relations between the ActionOccurrences connected by their corresponding ControlFlowOccurrences. In case of a ControlFlowOccurrence of a StrictControlFlow between two ActionOccurrences ao1 and ao2, we require that ao1 must be before ao2 or that ao1 meets ao2, i.e.

$$\forall cfo, cf, ao1, ao2 :$$
$$\mathsf{StrictControlFlow}(cf) \wedge \mathsf{ControlFlowOccurrence}(cfo)$$
$$\wedge\ \mathsf{occurrenceOf}(cfo, cf)$$
$$\wedge\ \mathsf{ActionOccurrence}(ao1)\ \wedge\ \mathsf{ActionOccurrence}(ao2)$$
$$\wedge\ \mathsf{from}(cfo, ao1)\ \wedge\ \mathsf{to}(cfo, ao2)$$
$$\Rightarrow\ \mathsf{before}(ao1, ao2) \vee \mathsf{meets}(ao1, ao2) \quad .$$

In wpo1 (Fig. 4), there is a single ControlFlowOccurrence *cfo* for which the premises of this axiom can be applied, i.e., the ControlFlowOccurrence between ao1 and ao2. Thus, we require that ao1 is before ao2 or ao1 meets ao2. As neither relation holds, the axiom is violated and wpo1 is invalid. In contrast wpo2 is valid, because ao1 before ao2.

The semantics of an OverlappingControlFlow is defined by an axiom analog to that of the StrictControlFlow, but where the time relation overlaps is allowed in addition to before and meets. Hence, wpo4 (Fig. 5) is invalid, because ao3 starts before ao1. wpo5 is valid because all axioms are respected.

So far, the semantics of ControlFlows has only been defined if their occurrences are from and to an ActionOccurrence. Other possibilities are reduced to this case. As an example, consider wpo3 in Fig. 4, which contains a path of two ControlFlowOccurrences from ao1 via a DecisionOccurrence to ao2. By definition, this path is equivalent to the most restrictive single ControlFlowOccurrence, i.e., an occurrence of a StrictControlFlowOccurrence in this example. Hence, wpo3 is equivalent to wpo2 and thus valid. In a similar way, wpo6 in Fig. 5 is equivalent to wpo5 and also valid. In general, the *most restrictive* one of a set of ControlFlows is the ControwFlow that allows *all* time relations allowed by each single ControwFlow, and *no further* time relations.

Note that wpo3 and wpo6 directly correspond to the example model in Fig. 2. wpo3 represents the case that a1 and a2 are executed, and it gives some valid (but still arbitrary) values for the start and end of the ActionOccurrences. wpo6

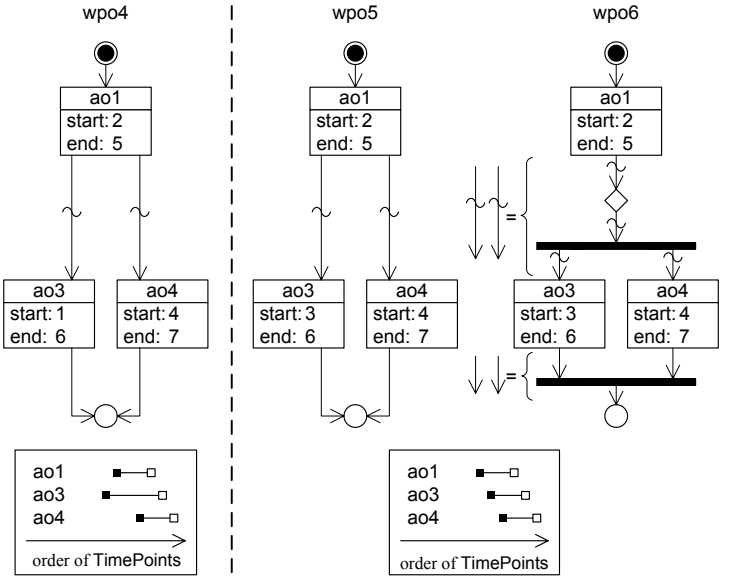


Fig. 5. Graphical depiction of three WorkProcessOccurrences. wpo4 is invalid because neither of the conditions ao1 overlaps ao3, ao1 before ao3, or ao1 meets ao3 holds. wpo5 and wpo6 are valid.

represents the case that a1, a3, and a4 are executed. The structure of both wpo3 and wpo6, excluding the time values, can be derived by means of the Petri net in Fig. 3[8]. Also, if we neglect the time values, wpo3 and wpo6 are the *only* WorkProcessOccurrences which can be generated by the Petri net.

The overall semantics of the example model is that it represents *all* concrete work processes described by wpo3 or wpo6 with arbitrary time values, provided the constraints induced by the axioms are respected.

5 Functional Modeling in WPML

In this section, we address the functional aspect of a work process. Each action in a work process typically serves a *function* which reflects the required input and the expected results. In a purely behavioral WPML model as discussed in the previous section, Actions produce or consume Objects[9] (see Fig. 6). To cover the functional aspect of a work process, Actions can be linked to the Functions

[8] The necessity to keep track of the transitions each token passes is one reason to use a more expressive Petri net variant than state-transition nets.

[9] The hasInputObject and hasOutputObject properties are simplifications to emphasize the analogy between the concepts on the behavioral side and those on the functional side. Actually, Actions and Objects are linked by ObjectFlows similar to the ControlFlows discussed in Sec. 4.

they fulfill, and Objects can be linked to States. Sections 5.1 and 5.2 deal with functional models of two different types work work processes. Finally, Sect. 5.3 elaborates the relation between behavioral and functional aspects in more detail.

In general, the functions that can appear in a work process model depend on the process type. For instance, within an operational process, the function of an action may be to change a reactor temperature to 80°C. To describe this function, concepts such as Reactor and SetPoint are required. Thus, subclasses of Function for different process types are introduced, such as the OperationalProcessStep for operational processes and the DesignStep for design processes. Fig. 6 shows typical function classes for both types of work processes.

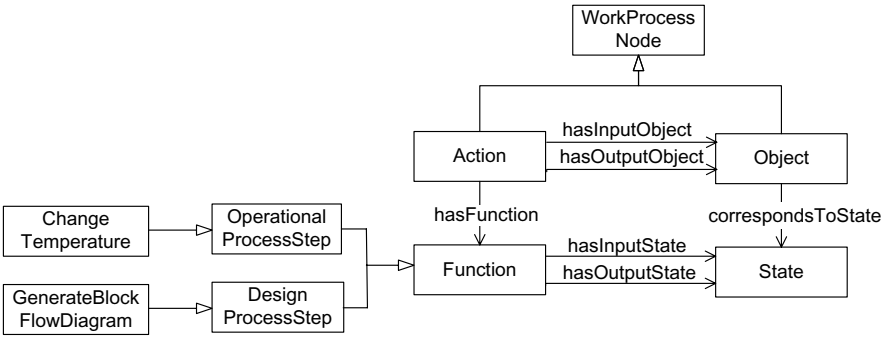


Fig. 6. Actions can be specified by Functions that serve to change an input State into an output State. The diagram shows exemplary function classes for design and operational processes.

The definition of adequate subclasses of Function allows to extend WPML with domain knowledge. As discussed in Sec. 2, such domain knowledge covers the function of work processes and the objects involved. This knowledge is represented as part of *OntoCAPE*, a comprehensive ontology for the CAPE (*computer-aided process engineering*) domain [53]. According to Gruber, an ontology is an explicit specification of a conceptualization, while a conceptualization can be seen as an abstract, simplified view of the world for a specific purpose [54]. By means of an ontology, the semantics of the modeling concepts is explicitly and formally specified.

Originally, OntoCAPE has been developed by our group to describe the artifacts created during the conceptual design of chemical processes. However, the extensible structure of the ontology allows to integrate concepts related to work processes seamlessly. That way, many concepts of OntoCAPE can be used to enrich the representation of the context of work processes. OntoCAPE is described comprehensively elsewhere [53,55,56]. In the following, we focus on the extensions developed for representing operational processes and design processes. These extensions are implemented as *partial models* of OntoCAPE.

5.1 Extensions for Operational Processes

During an operational process, a number of operational steps are conducted in different plant items. These operational steps support the chemical processes to achieve the desired chemical products. Accordingly, when modeling operational processes, modeling concepts must cover the operational steps (e.g. OpenValve or SetControllerParameter), the involved plant items (e.g. Instrumentation, Vessel or DistillationSystem), the production steps of chemical processes (e.g. Reaction or Distillation), as well as the materials processed or produced by the chemical process.

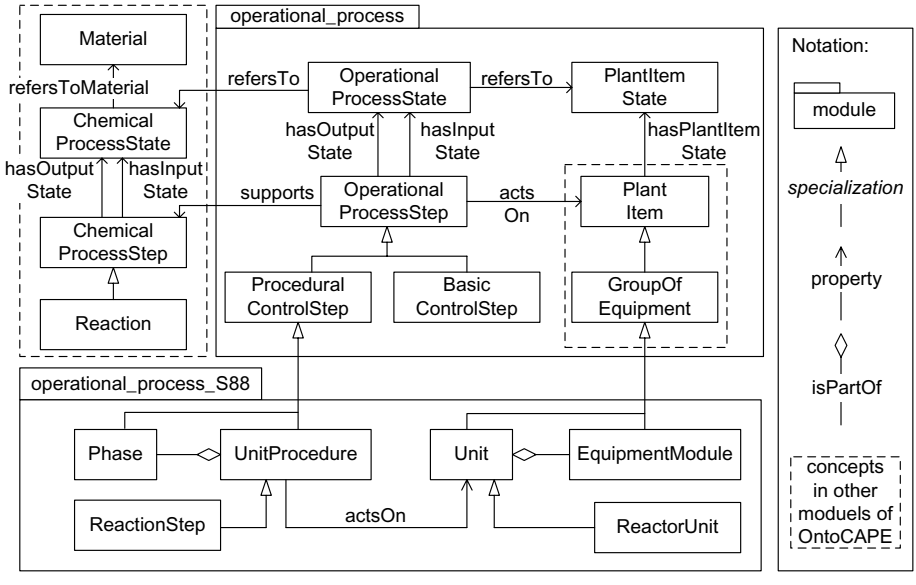


Fig. 7. The simplified partial model for operational processes

Moreover, concepts for a *structured* representation of chemical processes, operational processes, and plant items must be incorporated in order to permit descriptions on different levels of detail and granularity. In industrial practice, there are various ways to specify the hierarchies of the aforementioned entities. For instance, a very generic way to enable the structured representation of plant items is to specify a PlantItem as a PieceOfEquipment or a GroupOfEquipment. A more restricted alternative may involve concepts like ProcessCell, Unit, or SingleEquipment. Bearing in mind that various hierarchies may be needed when modeling operational processes, we propose an extensible modular structure for the partial models. The partial model for operational processes consists of the two modules shown in Fig. 7. While the module operational_process contains concepts for representing operational processes independent from any applications or standards, the module operational_process_S88 covers the concepts for

describing operational process according to the well established ANSI/ISA-S88 standard [33]. In the following, the modules involved in this partial model are explained in detail.

The module operational_process. This module is simple, yet contains essential concepts for describing operational processes. The most important concepts within this module are OperationalProcessStep and OperationalProcessState. An OperationalProcessStep acts on PlantItems and supports a ChemicalProcessStep. Intentionally, only a very generic specification for OperationalProcessStep is included in this module: an OperationalProcessStep can be a BasicControlStep or a ProceduralControlStep. A BasicControlStep is a simple elementary step such as to open a valve, to set the parameters for a controller, or to switch on a pump. A ProceduralControlStep, on the contrary, consists of several operational steps. For instance, a charging step may contain a set of basic operational steps such as to open a block valve or to start a pump. An OperationalProcessStep changes the OperationalProcessState which captures all relevant states during an operational process, including

- PlantItemStates such as the position of a valve or the level of a tank,
- ChemicalProcessStates such as the temperature or pressure of the processed material in a reactor, and
- ExecutionStates of an operational process step like *running*, *aborted* or *complete* (not shown in Fig. 7).

The module operational_process_S88. Because of the generic character of the operational_process module, additional modules for the further specification of operational processes based on certain standards or guidelines can be included with minor effort. For illustration, we choose the ANSI/ISA-S88 standard as an example for further specification of the operational_process module. This standard provides a solid foundation for a hierarchical representation of plant operational processes and plant items. The structure defined in ANSI/ISA-S88 is partially adopted in this work (cf. Fig. 7). The module operational_process_S88 enables to represent plant items on four levels—ProcessCell, Unit, EquipmentModule, and ControlModule—to refine the modules incorporated in OntoCAPE. Furthermore, the concept ProceduralControlStep is specified into Procedure, UnitProcedure, Operation, and Phase. Relations between these concepts are also defined. For instance, a UnitProcedure can only take place on a Unit.

The modules operational_process and operational_process_S88 provide sufficient expressiveness to describe operational processes on different levels of granularity and detail. For instance, the module operational_process defines that an OperationalProcessStep acts on PlantItems and that an OperationalProcessStep supports a ChemicalProcessStep. This can be further specified as, for instance, a ReactionStep supports a chemical process step of type Reaction carried out in a ReactorUnit. By decomposing the ReactionStep into some Phases, more details of the ReactionStep can be captured. Another possibility for the structured representation of plant items and operational processes has been reported in [57].

When necessary, this can be defined in an additional module to extend the operational_process module. When modeling an operational process, one of the extensions can be applied.

An adequate representation of operational processes can be used not only to improve the quality of plant operations, but also to support the design of the operational specifications, as shown in the next section. For a more comprehensive description of the partial model for operational processes as well as its application, see [58].

5.2 Extensions for Design Processes

Currently, both conceptual design processes and the design of operational process specifications are considered the design_process module. For representing the conceptual design process, we have adopted and extended the Douglas methodology [40]. The Douglas methodology uses a hierarchical approach where design decisions are taken on a set of predefined levels of abstraction. We have applied the Douglas methodology to represent conceptual design processes hierarchically. In a similar way, modeling concepts for design processes of operational specifications have been developed based on the hierarchy defined in the ANSI/ISA-S88 standard.

As typical objects involved in a design process, documents contain information about a chemical process or parts of the process. Representing the *content* of the involved documents is crucial for describing design steps with sufficient detail, because only then it is possible to describe the required information for carrying out a design step and the outcome of that design step. Using concepts defined in OntoCAPE, a chemical process system can be described under different aspects from the abstract process level to the concrete plant item level. Such a detailed model of domain objects provides an essential base for representing design steps on different levels of abstraction.

As shown in Fig. 8, the partial model for representing design processes contains several modules. Whereas the module design_process contains concepts for modeling design steps, the module document_model can be used to describe the documents created or modified by a design step. A number of further modules may be involved to represent the content of the documents. Two of these modules, chemical_process_Douglas and operational_process_S88 are shown in Fig. 8. While the module operational_process_S88 has been already introduced in Sec. 5.1, we discuss the other three modules in Fig. 8 in the following.

The module document_model. This module plays a crucial role to relate the design steps to the domain knowledge about objects within the domain of chemical engineering. On the topmost level, a Document is required, produced or modified as an input or an output of a DesignStep. The content of a Document can be described by concepts representing domain objects. As a further specification, a set of documents in chemical engineering, represented by the concept ChemEngDoc and its subclasses, has been identified and related to the corresponding content. For instance, a BlockFlowDiagram hasContent about ReactionSystems, an

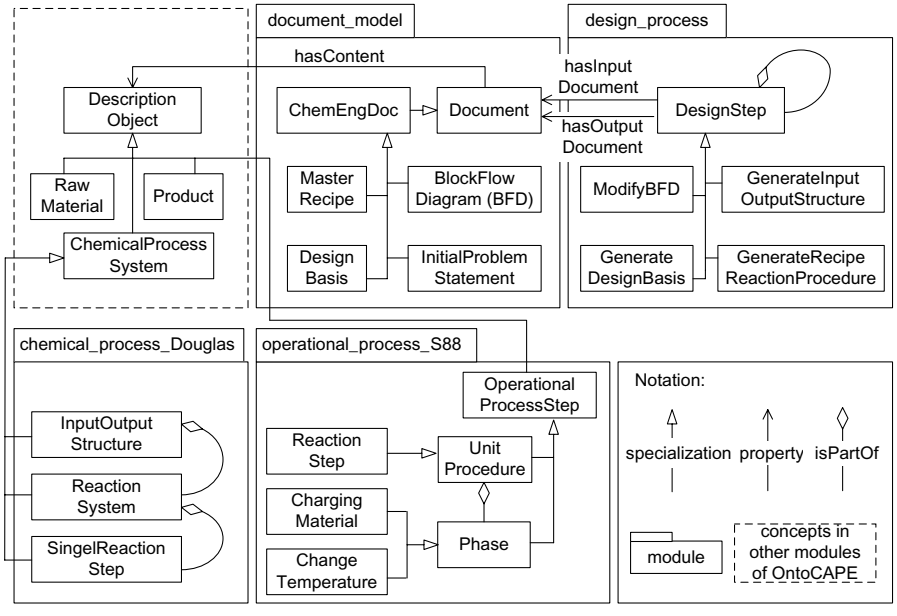


Fig. 8. The simplified partial model for representing design processes

InitialProblemStatement hasContent about RawMaterials and Products, or a MasterRecipe hasContent about a ReactionStep. The document_model module also captures a further aspect dealing with the management of document versions. More details about the document_model can be found in [59].

The module chemical_process_Douglas. This module includes a hierarchy based on the Douglas method to represent chemical process systems on different levels of abstraction. A chemical process system can be described on the following levels:

- The *process level* corresponds to the input-output structure in the Douglas hierarchy. On this level, only input and output materials are considered. Also recycle streams and purge streams can be investigated. Typical documents involved on this level are InitialProblemStatement and DesignProcessBasis.
- The *process system level* corresponds to the recycle structure of the Douglas hierarchy. On this level, reaction systems or separation systems are considered. Separation systems can be further classified as vapor separation systems, liquid separation systems, and flashes. Reaction system can be further specified into single reaction steps. Typical documents involved on this level are DesignProcessBasis and BlockFlowDiagram.
- The *standard unit level* is not included in the Douglas hierarchy, but is useful to describe standard unit operations and single reaction steps. Typical documents involved on this level are BlockFlowDiagram and ProcessFlowDiagram.

The module design_process. The topmost concept in this module is DesignStep which is further specified by typical design activities like *generate*, *modify* or *refine*, combined with the associated design product such as a block flow diagram or a master recipe. ModifyBlockFlowDiagram and GenerateMasterRecipe, for instance, are two specific DesignSteps. Representing design steps by combining design activities and the corresponding design products is a straightforward and intuitive way which helps to identify design steps in terms of the related design content. We intentionally avoid using typical terminology from the design research area like syntheses, analyze, induction, abduction etc. There are two main reasons for doing so. Firstly, design activities in industry are usually described by referring to the target product. For instance, *generate a flow sheet* is a typical design activity. Secondly, each design activity can be seen as an aggregate activity containing a synthesis step, an analysis step, and a decision step (cf. [46]). Hence, *synthesis*, *analysis*, and *decision* may be used as elementary building blocks for more expressive concepts, but are not sufficient for representing design processes.

For each DesignStep the required input and output documents are defined, both with the corresponding content. A DesignStep can be further decomposed into other DesignSteps. For the conceptual design process, design steps are introduced according to the levels defined in the module chemical_process_Douglas and the involved documents in chemical engineering. A similar approach is used for the design process of operation specifications. The modules operational_process_S88 is used to describe the content of the design steps in a plant operation design process. For illustration, two simple examples are given in the following.

Example 1: A design step during plant operation specification. A design step of type GenerateRecipeReactionStep requires a certain MasterRecipe (see Fig. 9). After this design step, the second version of this MasterRecipe is created. The second version contains information about a ReactionStep for the reactor R3. This ReactionStep contains two Phases: Charging reactant 1 of type ChargingMaterial and Heating of type ChangeTemperature.

Example 2: A design step in conceptual design. To develop a chemical process for producing benzene by hydrodealkylation (HDA) of toluene, a number of design steps are to be carried out (see Fig. 10). This example describes a design step of type GenerateInputOutputStructure (IOS) which requires a document of type InitialProblemStatement. This document contains a statement of the involved raw materials (i.e., Toluene and Hydrogen) and the desired products (i.e., Benzene and Methane). In addition, information about the ChemicalReaction of interest (i.e. the hydrodealkylation of Toluene) is also given in the InitialProblemStatement. The result of this design step is a report of type DesignBasis, which has content about the InputOutputStructure of this HDA process.

These two simple examples show the strength and necessity to incorporate detailed models of domains objects such as ChemicalReaction or RawMaterial when modeling design processes. Knowledge about both, the design processes

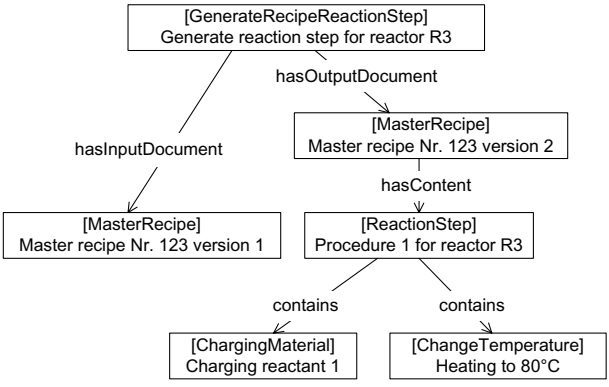


Fig. 9. Example of a design step for plant operation on the instance level

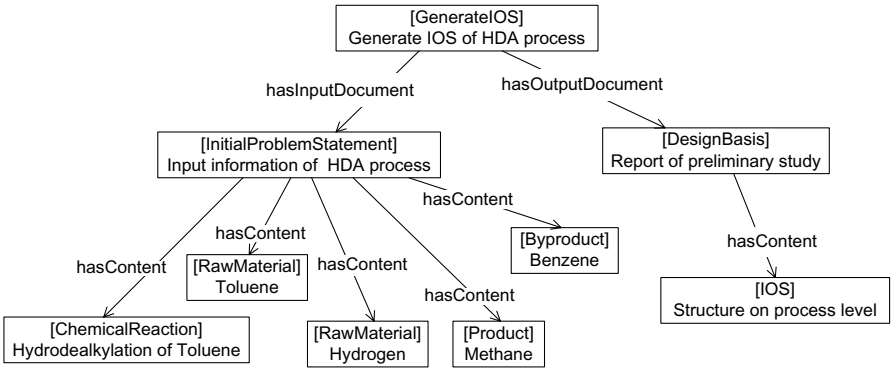


Fig. 10. Example of a conceptual design step on the instance level

and the involved objects, must be encompassed from the coarse-grained level to the fine-grained level to represent design steps with sufficient information.

5.3 Integrating Process Behavior and Function

So far, we have discussed the behavioral and the functional aspect of work processes independent from each other. This section is meant to illustrate the integration of the two aspects. As a simple (and incomplete) example, consider the work process model in Fig. 11. The left part of the figure shows part of a design process for HDA in the concrete syntax of WPML that has been adopted from UML activity diagrams. In the right part, the same design process is represented as an object diagram. The two Actions of the process are linked by a ControlFlow that captures the behavioral aspect of the process. Also, to each Action, an instance of a concrete Function class is assigned to represent the functional aspect.

Note that in the concrete syntax, it is sufficient to display the name of this function class. As for the functional aspect, it could be further specified as shown in the examples given in Sec. 5.2.

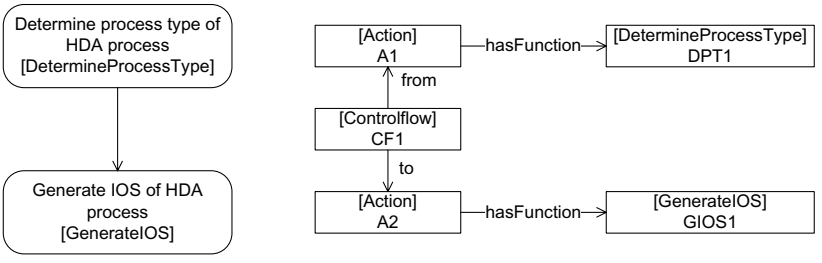


Fig. 11. Example of a conceptual design process on the instance level. Both behavioral and functional aspects are captured.

6 Implementation and Tool Support

Both WPML and its extensions are implemented in OWL [47]. The formal specification of process behavior is additionally supported by Petri nets and axioms in first-order logics. WPML and the extensions can be seen as meta-models which provide the building blocks for work process models on the instance layer. We use the OWL editor Protégé [60] to implement the meta-models.

For the creation of work processes on the instance layer in a user-friendly manner, our research group has developed the Work Process Modeling System (WOMS+). WOMS+ provides a set of intuitive symbols for the WPML modeling concepts. In addition, meta-models for different types of work processes (e.g., the module for operational processes) can be imported into WOMS+, such that the classes and properties defined in these models are offered to the user, who can select among them for the formal specification of the elements in a WPML instance model. Moreover, further use of the models created by means of WOMS+ is also supported by an automated export into certain target formats of applications like the Aspen Batch Process Developer or a Petri net simulators.

7 Conclusions and Open Issues

This contribution motivates and presents the WPML language, a generic modeling language suitable for different types of work processes, which allows for further extensions specific to certain types of work processes. Extensions for design and operational processes have been discussed.

The WPML language supports the formal representation of both, the behavioral and functional aspects of a work process. Domain knowledge can explicitly be incorporated on different levels of abstraction. Not only the work process itself, but also the objects involved are covered with sufficient details. Models

created with this language can easily be translated in different format such that *one* model can be used by different applications. This strategy supports the reuse of a work process model which captures the knowledge of an organization and thus increases engineering productivity.

Currently, cooperation with several industrial partners is in progress to achieve a practical evaluation of the modeling framework. These empirical studies aim at verifying the usability of the tool as well as the usability and expressiveness of WPML.

Acknowledgments. The authors acknowledge financial support from the German National Science Foundation (DFG) under grant MA 1188/29-1.

References

1. Sharp, A., McDermott, P.: Workflow Modeling: Tools for Process Improvement and Application. Artech House, Norwood (2001)
2. Davenport, T.H.: Process Innovation. Harvard Business School, Boston (1993)
3. Workflow Management Coalition (WfMC): Terminology & Glossary, Report No. WFMC-TC-1011 (1999), http://www.wfmc.org
4. Nagl, M., Marquardt, W.: Collaborative and Distributed Chemical Engineering: from Understanding to Substantial Design Process Support; Results of the IMPROVE Project. LNCS, vol. 4970. Springer, Heidelberg (2008)
5. Theißen, M., Hai, R., Marquardt, W.: Design Process Modeling in Chemical Engineering. J. Comput. Inf. Sci. Eng. 8(1), 011007 (9 pages) (2008)
6. Theißen, M., Hai, R., Morbach, J., Schneider, R., Marquardt, W.: Scenario-based Analysis of Industrial Work Processes. In: Nagl, M., Marquardt, W. (eds.) Collaborative and Distributed Chemical Engineering. LNCS, vol. 4970, pp. 433–450. Springer, Heidelberg (2008)
7. Killich, S., Luczak, H., Schlick, C., Weienbach, M., Wiedenmaier, S., Ziegler, J.: Task Modelling for Cooperative Work. BIT 18(5), 325–338 (1999)
8. Eggersmann, M., Kausch, B., Luczak, H., Marquardt, W., Schlick, C., Schneider, N., Schneider, R., Theißen, M.: Work Process Models. In: Nagl, M., Marquardt, W. (eds.) Collaborative and Distributed Chemical Engineering. LNCS, vol. 4970, pp. 126–152. Springer, Heidelberg (2008)
9. Object Management Group (OMG): Unified Modeling Language, Version 2.0 (2005), http://www.omg.org/uml
10. Theißen, M., Marquardt, W.: Decision Process Modeling in Chemical Engineering Design. In: 17th European Symposium on Computer Aided Process Engineering, pp. 383–388 (2007)
11. Marquardt, W., Theißen, M.: Integrated Modeling of Work Processes and Decisions in Chemical Engineering. In: Schlick, C.M. (ed.) Industrial Engineering and Ergonomics, pp. 265–279. Springer, Heidelberg (2009)
12. Aspen Technology, http://www.aspentech.com/products/aspen-batch-plus.cfm
13. Bayer, B., Marquardt, W.: Towards Integrated Information Models for Data and Documents. Comput. Chem. Eng. 28, 1249–1266 (2004)
14. Brandt, S., Morbach, J., Miatidis, M., Theißen, M., Jarke, M., Marquardt, W.: An Ontology-Based Approach to Knowledge Management in Design Processes. Comput. Chem. Eng. 32(1-2), 320–342 (2008)

15. Object Management Group (OMG): Model Driven Architecture, http://www.omg.org/mda
16. Petri, C.A.: Kommunikation mit Automaten. Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik (1962)
17. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Basic Concepts, vol. 1, 2nd corrected printing. Springer, Berlin (1997)
18. Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities. In: Liggesmeyer, P., Pohl, K., Goedicke, M. (eds.) Software Engineering. LNI, vol. 64, pp. 117–128. Gesellschaft für Informatik (2005)
19. van der Alst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Inform. Syst. 30(4), 245–275 (2005)
20. National Institute of Standards and Technology (NIST): The Process Specification Language (PSL): Overview and Version 1.0 Specification, Report No. NISTIR 6459. Gaithersburg, MD (2000)
21. Bock, C., Gruninger, M.: PSL: A Semantic Domain for Flow Models. SoSyM. 4, 209–231 (2005)
22. Object Management Group (OMG): Business Process Modeling Notation (BPMN) (2006), http://www.omg.org
23. BEA Systems, IBM, Microsoft, SAP AG, Siebel Systems: Business Process Execution Language for Web Services, version 1.1 (2003), http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/
24. Workflow Management Coalition (WfMC): XPDL 2.0 Specification (2005), http://www.wfmc.org
25. White, S.A.: XPDL and BPMN. In: Workflow Handbook 2003, pp. 221–238. Future Strategies Inc., Lighthouse Point (2003)
26. Mayer, R.J., Painter, C.M.K., de Witte, P.S.: IDEF Family of Methods for Concurrent Engineering and Business Re-Engineering Applications, http://www.idef.com/
27. National Institute of Standards and Technology (NIST): Integrated Definition for Function Modeling (IDEF0), Report No. NISTIR 183. Gaithersburg, MD (1993)
28. Batres, R., Naka, Y.: Process Plant Ontologies Based on a Multi-dimensional Framework. In: Malone, M.F., Trainham, J.A., Carnahan, B. (eds.) Proceedings of the Fifth International Conference on Foundations of Computer-Aided Process Design, pp. 433–437 (2000)
29. Knowledge Systems Laboratory: Ontolingua, http://www.ksl.stanford.edu/software/ontolingua
30. VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik: Classification and Evaluation of Description Methods in Automation and Control Technology, VDI/VDE 3681 (2005)
31. International Electrotechnical Commission (IEC): Programmable controllers – Part 3: Programming languages, IEC 61131-3 Ed. 2.0 (2003)
32. VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik: Formalised Process Descriptions, VDI/VDE 3682 (2005)
33. International Society of Automation (ISA): Batch control Part 1: Models and terminology, ANSI/ISA-S88.01-1995 (1995)
34. Viswanathan, S., Johnsson, C., Srinivasan, R., Venkatasubramanian, V., Ärzen, K.E.: Automating Operating Procedure Synthesis for Batch Processes: Part I. Knowledge Representation and Planning Framework. Comput. Chem. Eng. 22(11), 1673–1685 (1998)
35. Gabbar, H.A., Aoyama, A., Naka, Y.: Recipe Formal Definition Language for Operating Procedures Synthesis. Comput. Chem. Eng. 28(9), 1809–1822 (2004)

36. Chandrasekaran, B.: Design Problem Solving: A Task Analysis. AI Mag. 11(4), 59–71 (1990)
37. Gero, J.S., Kannengiesser, U.: A Function-Behavior-Structure Ontology of Processes. AI EDAM 21(4), 379–391 (2007)
38. Stone, R.B., Wood, K.L.: Development of a Functional Basis for Design. J. Mech. Des. 122(4), 359–370 (2000)
39. Kitamura, Y., Riichiro, M.: Ontology-Based Systematization of Functional Knowledge. J. Eng. Des. 15(4), 327–351 (2004)
40. Douglas, J.M.: A Hierarchical Decision Procedure for Process Synthesis. AICHE J. 31, 353–362 (1985)
41. Douglas, J.M.: Conceptual Design of Chemical Processes. McGraw-Hill, New York (1988)
42. Gorti, S.R., Gupta, A., Kim, G.J., Sriram, R.D., Wong, A.: An Object-Oriented Representation for Product and Design Processes. Comput.-Aided Des. 30(7), 489–501 (1998)
43. Bañares-Alcántara, R.: Design Support Systems for Process Engineering – I. Requirements and Proposed Solutions for a Design Process Representation. Comput. Chem. Eng. 19(3), 267–277 (1995)
44. Bayer, B.: Conceptual Information Modeling for Computer Aided Support of Chemical Process Design. Fortschritt-Berichte VDI: Reihe, vol. 3(787). VDI-Verlag, Düsseldorf (2003)
45. Eggersmann, M.: Analysis and Support of Work Processes within Chemical Engineering Design Processes. Fortschritt-Berichte VDI: Reihe, vol. 3(840). VDI-Verlag, Düsseldorf (2004)
46. Eggersmann, M., Gonnet, S., Henning, G., Krobb, C., Leone, H., Marquardt, W.: Modeling and Understanding Different Types of Process Design Activities. Lat. Am. Appl. Res. 33, 167–175 (2003)
47. World Wide Web Consortium: OWL Web Ontology Language. Reference. Recommendation (2004), http://www.w3.org/TR/owl-ref/
48. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
49. Clark & Parsia.: Pellet OWL Reasoner (2010), http://clarkparsia.com/pellet/
50. Theißen, M., Hai, R., Marquardt, W.: A Framework for Work Process Modeling in the Chemical Industries. Submitted to Comput. Chem. Eng. (2009)
51. Horridge, M., Patel-Schneider, P.F.: Manchester Syntax for OWL 1.1. In: Clark, K., Patel-Schneider, P.F. (eds.) OWL: Experiences and Directions 2008 DC, Fourth International Workshop, Washington, DC (2008)
52. Allen, J.F.: Maintaining Knowledge about Temporal Intervals. Comm. ACM 26(11), 832–843 (2009)
53. Marquardt, W., Morbach, J., Wiesner, A., Yang, A.D.: OntoCAPE - A Re-Usable Ontology for Chemical Process Engineering. Springer, Berlin (2010)
54. Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. Knowl. Acquis. 5(2), 199–220 (1993)
55. Morbach, J., Yang, A.D., Marquardt, W.: OntoCAPE - A Large Scale Ontology for Chemical Process Engineering. Eng. Appl. Artif. Intel. 20(2), 147–161 (2007)
56. Morbach, J., Wiesner, A., Marquardt, W.: OntoCAPE 2.0 - A (Re)usable Ontology for Computer-Aided Process Engineering. Comput. Chem. Eng. 33, 1546–1556 (2009)
57. Aoyama, A., Yamadai, I., Batres, R., Naka, Y.: Multi-Dimensional Object Oriented Approach for Automatic Generation of Control Recipes. Comput. Chem. Eng. 24(2-7), 519–524 (2000)

58. Hai, R., Theißen, M., Marquardt, W.: An Ontology Based Approach for Operational Process Modeling. Submitted to Advanced Engineering Informatics (2010)
59. Morbach, J., Hai, R., Bayer, B., Marquardt, W.: Document models. In: Nagl, M., Marquardt, W. (eds.) Collaborative and Distributed Chemical Engineering. LNCS, vol. 4970, pp. 111–125. Springer, Heidelberg (2008)
60. Stanford Center for Biomedical Informatics Research: The Protégé ontology editor and knowledge acquisition system, http://protege.stanford.edu

Integration Tools for Consistency Management between Design Documents in Development Processes

Simon M. Becker[1] and Anne-Thérèse Körtgen[2]

[1] Inform GmbH,
Pascalstrasse 23, D-52076 Aachen, Germany
Simon.Becker@inform-ac.com
[2] RWTH Aachen University, Department of Computer Science 3,
Ahornstrasse 55, D-52074 Aachen, Germany
koertgen@se-rwth.de
http://www.se-rwth.de/staff/koertgen

Abstract. Development processes in engineering disciplines are inherently complex. Throughout the development process, the system to be built is modeled from different perspectives and on different levels of abstraction in multiple documents. They are related by manifold dependencies and need to be maintained mutually consistent with respect to these dependencies. In addition, development processes are highly incremental and iterative. Thus, tools are urgently needed which assist developers in maintaining consistency between inter-dependent documents. These tools have to operate incrementally and need to support user interactions, as the effects of changes cannot always be determined automatically and deterministically. At the Department of Computer Science 3 at RWTH Aachen University, triple graph grammars (TGG) have been invented as a formal approach to handling integration problems. During multiple research activities at the department and many other research groups, TGGs have been used as a basis to design algorithms and modeling formalisms and have been further elaborated. At the department they were implemented and enhanced in different ways for the support of rapid integration tool construction for real-world interactive development processes. This paper gives an insight into specification, algorithms, and tool construction for interactive, incremental integration and shows the application within the design process of a chemical plant executed in a commercial product.

1 Introduction

Development processes in engineering disciplines are inherently complex. During the development process, the product to be developed is *modeled* from different perspectives, on different levels of abstraction, and with different intents in multiple documents.

Documents are related by manifold *dependencies* and can get inconsistent to dependent documents if they change. For example, the design of a software system depends on its requirements, and the implementation depends on the design. There exist rules which constrain the elements of a document due to the dependency to the elements of that document to which it is dependent, such as the elements of the design are constrained due to the dependency to the requirements. The constraints can be expressed

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 683–718, 2010.

by relations between elements of dependent documents. If these constraints are satisfied, a document is said to be consistent with its dependent document (*inter-document consistency*).

Development processes may be viewed as multi-phase *transformation processes* from the initial problem statement to the final solution. However, this is a simplified view suggesting a waterfall-like process, where each phase is entered only when the preceding phase has been completed. In contrast, current development processes are highly *incremental* and *iterative*. Furthermore, development does not always proceed in forward direction (*forward engineering*). Rather, it may also involve activities working in backward direction (*reverse engineering*). Combining forward and reverse engineering results in *round-trip engineering*, where developers opportunistically mix both modes of development.

Maintaining inter-document consistency is a demanding task which requires sophisticated tool support. In this paper, we will subsume all kinds of tools for maintaining inter-document consistency under the notion of *integration tool*. These tools may be classified as follows: A *transformation tool* processes a source document and transforms it into a target document. A *consistency analysis tool* takes inter-dependent documents and checks inter-document consistency. A *hyperlink tool* creates and maintains links between elements of inter-dependent documents. A *browsing tool* traverses inter-document links.

Transformation tools have been studied extensively. For example, consider the *Model Driven Architecture* [1] initiative launched by the OMG. Often, transformation tools of this kind are run in batch mode. Batch transformers are not always adequate. Rather, the participants of the design processes have to make deliberate *design decisions* which balance requirements such as efficiency, adaptability, costs, etc. Therefore, the design process is performed *interactively*. Furthermore, design usually proceeds *incrementally* and *iteratively*. As a consequence, integration tools which assist in maintaining consistency between requirements and design need to operate both interactively and incrementally. Moreover, tool support has to provide for *traceability*, i.e., each element of the design has to be traced back to its originating elements in the requirements.

In this paper, we focus on incremental and interactive integration tools for inter-document consistency maintenance which were developed at the Department of Computer Science 3 chaired by Prof. Nagl. The development of integration tools has passed through multiple phases. The starting point of all realized integration tools at the department was the IPSEN project in 1982. It aimed at *a-priori* integration of languages and tools for the whole software development process via the *I*ntegrated and Incremental Software *P*roject *S*upport *En*vironment (IPSEN). Within the IPSEN-88 prototype [2,3] integration tools were coded manually, they worked *unidirectionally* and *incrementally*. Only $1:1$ relationships between *increments* (elements of document) were supported.

In the IPSEN-90 prototype [4] relationship representing *links* were stored in a separate document, the *integration document*, and, therefore, $n:m$-relationships could be supported as well as browsing between inter-dependent documents. Storing links in a separate data structure allows *a-posteriori integration* of existing tools, although not within the scope at that time. In the case of heterogeneous tools, each tool maintains its own data base, which may not be extensible by external applications. Furthermore, in

IPSEN-90 *bidirectional* transformation was introduced. With the third IPSEN prototype [5] *interactive* integration was supported, but reaction on changes was hard coded. The fourth and last IPSEN prototype [6] then was designed for *a-posteriori* integration and based on *triple graph grammars* (TGG) [7]. TGG rules specified synchronous development of two independent documents and the integration document. From that TGG rules a set of asynchronous transformation rules were derived manually into program code.

Within the Collaborative Research Center (CRC) 476 IMPROVE [8] initiated and headed by Prof. Nagl, integration tools were generalized to support also other development processes. The CRC's scope of application were chemical engineering design processes to show applicability of the generic concepts. Current integration tools are built on an *integrator framework* [9] which is augmented with transformation rules derived automatically from TGG rules. To allow for user interaction during integration, transformation rules are split into *graph matching* and *transformation* rules which are executed by a specially designed algorithm. Originally, TGGs have been used for incrementally transforming new increments from one document to the other (and vice-versa), while persistently linking source and target. In addition, changes on already transformed increments can be propagated by the application of repair actions which are derived automatically from TGG rules at runtime [10]. With the use of wrappers nearly all graph-based documents can be connected to the integration tools. In this paper, the approach of integration tools realized within the CRC representing the results of research of the last decades are presented in more detail.

The next section aims to give an overview of integration tools, the scenarios they support, the way they are constructed, and the way how they work. In Section 3, we explain the underlying concepts and algorithms of integration tools which are based on graphs and graph transformations. In that section, we are approaching the subject in a more theoretical way, mostly referring to a proof-of-concept realization based on PROGRES [11]. The concepts learned there have been transferred into today's integration tools which are realized for real-world industrial application. Those tools are presented in Section 4, also highlighting investigations for making integration tools more user-friendly and of providing support tools for modeling the TGG rules. In Section 5, we give a brief overview of related work being performed in this research area, and finally draw a short conclusion in Section 6.

2 Overview of Integration Tools

Using an example from a chemical engineering design process, it is explained how interactive and incremental integration tools can be applied in development processes (Section 2.1). Section 2.2 presents a set of characteristics derived from the scenario. The system architecture of an integration tool is presented in Section 2.3 and in Section 2.4 it is explained on a high level how integration is performed.

2.1 Scenario and Motivation

The research reported in this paper has been carried out within the IMPROVE project [12,8], which is concerned with models and tools for design processes in *chemical engineering*. In this section, we present a small example which illustrates key features

of incremental and interactive integration tools. This example is drawn from chemical engineering, but we could also have chosen an example from another engineering discipline (e.g., software engineering).

In chemical engineering, the *process flow diagram* (PFD) acts as a central document for describing the chemical process. The PFD is refined iteratively so that it eventually describes the chemical plant to be built. Simulations are performed in order to evaluate design alternatives. Simulation results are fed back to the PFD designer who annotates the PFD with flow rates, temperatures, pressures, etc. Thus, information is propagated back and forth between PFDs and *simulation models*. Although the PFD plays the role of a master document, it may also happen that a simulation model is created first and the PFD is derived from the simulation model (reverse engineering).

In general, PFDs and simulation models are created by different users at different times with the help of respective tools. In a cooperation with an industrial partner, we studied the coupling of COMOS [13], an environment for chemical engineering which in particular offers a PFD editor, and Aspen Plus [14], an environment for performing steady-state and dynamic simulations.

Unfortunately, the relationships between PFDs and simulation models are not always straightforward. Different kinds of simulation models are created for different purposes. Often, simulation models have to be composed from pre-defined blocks which in general need not correspond 1:1 to structural elements of the PFD. Thus, maintaining consistency between PFDs and simulation models is a demanding task requiring sophisticated tool support.

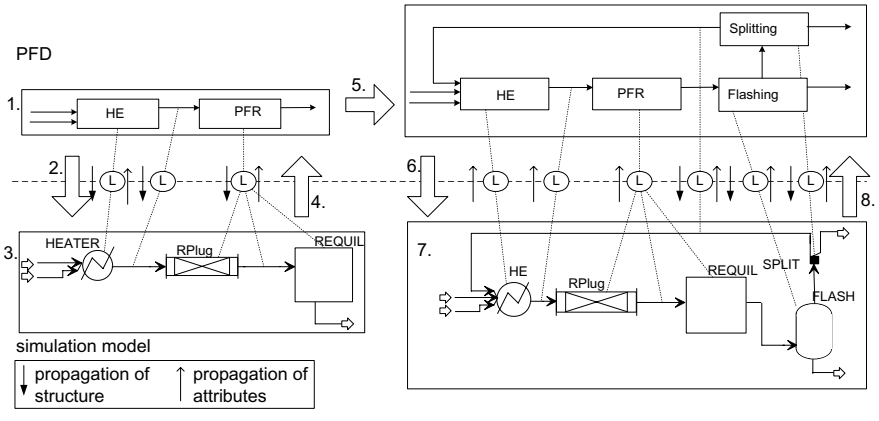


Fig. 1. Integration between PFD and simulation model

Figure 1 illustrates how an incremental integration tool assists in maintaining consistency between PFDs and simulation models. The chemical process taken as example produces ethanol from ethen and water. PFD and simulation model are shown above and below the dashed line, respectively. Objects of both documents (increments) are connected by links which are drawn on the dashed line and which are persistently stored

in an *integration document* after having been installed. Dotted lines indicate the increments participating in a link.

The figure illustrates a tiny part of the example design process consisting of 8 steps:

1. An initial PFD is created by the designer using COMOS PT. This diagram is usually incomplete, describing only a part of the chemical process (here heating of substances HE, reaction in a plug flow reactor PFR).
2. The designer wants to evaluate the design by a simulation, or he wants a simulation expert to do this, before completing the PFD. Therefore, the integrator is used to transform the PFD forward to a simulation model for Aspen Plus. While the heating step is transformed automatically into a Heater (assuming there is only this alternative), the engineer has to select one of different alternatives for mapping the PFR. He decides to use a cascade of two blocks to simulate the reaction.
3. Then the simulation is performed in Aspen Plus, which may include a further elaboration of the simulation model, not shown in Figure 1.
4. The results of the simulation are then used to annotate the PFD. So, here we have a backward transformation of attribute values from the simulation to the PFD.
5. The designer now decides to extend the PFD by new process steps (Flashing and Splitting).
6. The integrator is used again in forward mode. Here now two simulation blocks (SPLIT and FLASH) are inserted into the simulation model to match the changes of the PFD. It is important and necessary that the simulation model part of the transformation step 2 and its further extensions in step 3 remain unchanged.
7. Now, the simulation is performed again. We have further results, but also changed results corresponding to those parts of the simulation model already existing.
8. The simulation results are again propagated backward to the PFD.

We can see from this simple example that a design process is performed iteratively and incrementally, always having different documents in mind, here PFD and simulation model. So, we also have steps where we go back and refine, try something else, or even study a new variant. A user's changes may also affect objects which were already transformed successfully by the integration tool and which have a corresponding structure in a dependent document. This holds true also in maintenance where we usually modify larger portions of a design document. These changes then have to be propagated, until all documents are again consistent to each other.

2.2 Characterization of Document Relations and Tool Functionality

Regarding the above example, we derive the following requirements for integrators:

Functionality. In order to manage inter-document consistency, an integrator must deal with links between objects of source and target documents which, in general, are $m : n$. The tool has to install the links. The user would never do it manually, as the profit is too small.

Operation mode. The integrator must operate incrementally rather than batch-wise. It has to propagate changes between interdependent documents, such that only actually affected parts are propagated. As a consequence, manual elaborations (as above of the simulation model) do not get lost. Only in those cases batch converters are possible, where the target document can fully be generated.

Direction. In general, integrators have to work bi-directionally. Changes of the source document have to be propagated forward, those of the target document backward.

Interaction. In simple scenarios an automatic execution mode is possible, in most scenarios user interaction is required to select one of multiple design decisions, to react on decisions of others, or to resolve conflicts.

Time of activation. Only in single-user applications eager propagation of changes is useful; the user gets immediate reactions about consequences of changes. In multi-user scenarios, which is the regular case of industrial development, deferred propagation of changes is required after these changes are quality-assured.

A-posteriori integration. Integrators should work with heterogeneous tools of different vendors, which are used to support the elaboration of source and target documents. So, they have to access these tools and their data.

Integration knowledge. An integrator is driven by underlying knowledge, which patterns may relate to each other. These patterns on object level are defined by rules. There must be support for defining and applying these rules. In those cases where no appropriate rule exits, it must be possible to install or correct links in a manual mode.

Traceability. Integrators must record the rules having been applied. This way, the user can reconstruct which rules have been applied and when in the integration process.

Adaptability. The construction of integrators or the integrator itself must be adaptable to application domains, specific habits etc. Adaptability is achieved by defining suitable rules and controlling their application. Eventually, it must be possible to modify the rule base on the fly.

Not all these requirements have to be met by all integrators. In those situations, where the target document is completely generated, incrementality is not needed. In situations where the rule base is unambigious, no user actions are needed. We also have implemented such simple integrators. They have been realized in a simpler way than complex integrators, which use the integrator framework sketched below.

Figure 2 shows an example rule. It is formulated in the user view, i.e. only those items are presented the user needs for the application of the rule. The rule says that a PFR of the PFD may have a refinement in form of a sequence of a RPlug and a REQUIL in the simulation model. This rule was applied in the scenario of Figure 1. Figure 2 also shows the correspondence, i.e. the semantic link between the items of the PFD and the two items of the simulation model. The link between the object of the PFD and the two objects of the simulation model is stored persistently, as it may later be used for maintenance or other purposes.

If we look at the example above we derive the following general characterization: Entities of two documents (source document - target document which are dependent on each other) are to be connected by fine-grained dependency relations (semantic links). Dependency between two documents in the design process can have different semantics: (a) requirements for the chemical process - a process structure (PFD) compatible

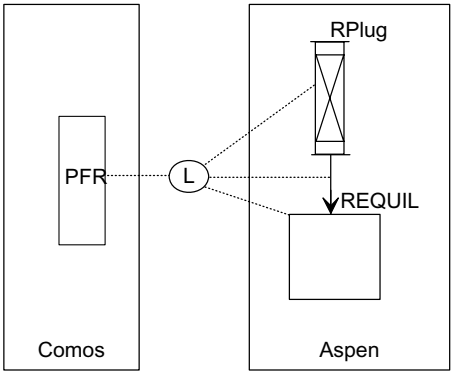


Fig. 2. Example of a correspondence rule

with these requirements, (b) coarse structure of a plant (in form of an overview PFD) - followed and detailed (by other refined PFD documents), (c) structure of a plant (as a PFD) - detailed and in a corresponding structure (in form of a piping and instrumentation diagram P&ID), (d) a structure (in form of a PFD) - to be analyzed and evaluated by another one (simulation diagram), and also other and further dependency forms. In this paper we are regarding relations of kind (d) in Figure 1 and relations of kind (c) in the following two sections. The tools we describe in this paper do not distinguish between different kinds of dependency relations, i.e. they are on a syntactic level. In all cases, structures of source and target document are related to each other. The corresponding structures, however, are different according to the variety of semantics of dependency relations.

2.3 System Overview

The coarse-grained system architecture of the integrator for our evaluation scenario (cf. Section 2.1) is depicted in Figure 3. Integrators are framework-based, i.e. most functionality is reused by all integrators and combined with integrator-specific code. In fact, the framework is based on .NET and was continuously developed [15, 16, 17]. This overview reflects all parts of the graph-based approach described in the next section.

Our approach is model-based, in particular graph models are considered. Graphs consisting of nodes and edges are well suited for representing complex data with manifold relationships in a natural way. In our approach, we use graphs to represent the application documents and the integration document placed in between these documents. Existing applications and their documents are connected to the framework using *tool wrappers* that provide a graph interface on their data. In our example, these tools are *Comos PT* and *Aspen Plus*. The corresponding wrappers both use the applications' COM interfaces to access their API. The relationships between the documents' data are stored in an additional *integration document*. It is serialized as XML file, but kept in memory during runtime of the integrator providing optimized access to its content, e.g., via indexes.

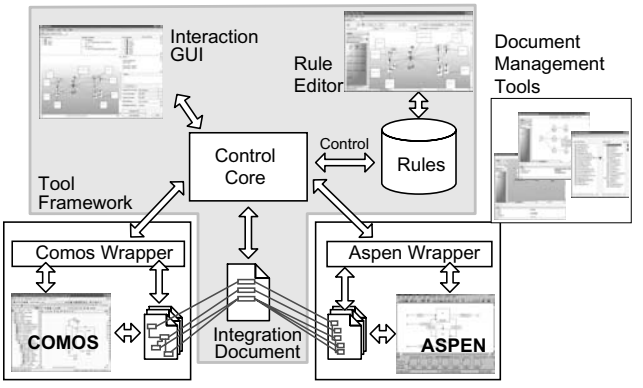


Fig. 3. System architecture of framework-based integrator

Although the manual definition of links is possible, in most cases integrators use *integration rules* to propose where to create a link. So, framework-based integrators are controlled by integration rules. The integration rules have the semantics of graph transformations which define how to relate graph patterns found in the documents by creating link structures in the integration document and how to create corresponding patterns to patterns in one document that are still missing in the other document. The concept of TGG on which integration rules are based is presented in more detail in Section 3.1.

The framework also includes tool support (upper right corner) for *rule modeling* which is based formally on the UML [18,19] (i.e., the graph grammar formalism is not exposed to domain experts). Based on the graph views the integration is performed by means of graph transformations which are defined by the UML integration rules. They are interpreted at runtime by the the *integrator core*, which is the main component of the framework. This allows for extending the integration rule set at runtime without having to recompile the integrator. Thus, the integrator can 'learn' new rules from manual interaction when a complete rule set is not available in advance [19].

This rule interpreter is based on a graph transformation engine that is part of the integrator core, too. This engine is kept much more light-weight than, e.g., PROGRES, as only very simple graph transformations (i.e., only adding and deleting nodes and edges and attribute transfers have to be done, pre-conditions are not supported) have to be executed. This is also possible because all pattern matching is done starting from dominant increments, in most cases only locally traversing the graph avoiding global pattern matching. Thus, the—in theory—high complexity of pattern matching does not affect the integrators' performance. The rule-independent graph transformation rules of the integration algorithm are manually hard-coded into the integrator core, making use of the optimized storage of links in the integration document. For further reading please be referred to [16,9].

During the integration process, integration rules may stand in conflict. Informally, two rules stand in conflict if the execution of one rule disables the execution of the

competing rule. Conflicts are presented to the user, who performs a selection among the conflicting rules. The GUI is presented in more detail in Section 4.2.

2.4 Overall Integration Algorithm

Here, we sketch the overall integration algorithm of our interactive, incremental, and bidirectional integration tools which execute forward and backward transformations and correspondence analysis as indicated in Section 2.1. It is shown in Figure 4.

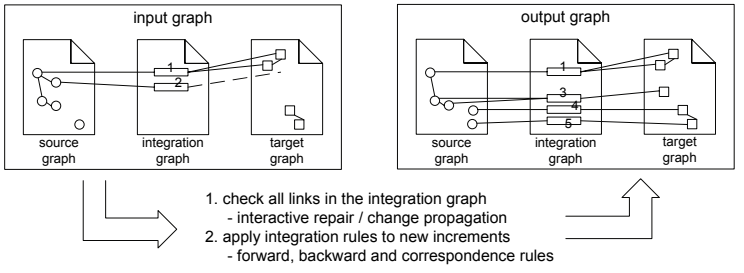


Fig. 4. Overall integration algorithm

The input and output of the algorithm is an overall graph that contains graph representations of source, target, and integration subgraphs. Thus, graph edges between nodes belonging to different subgraphs can be handled as normal graph edges. When the algorithm is invoked, source and/or target graph contain some edges and nodes, with some of the nodes already being connected to links in the integration graph, if the algorithm is not invoked for the first time. Another input to the algorithm is a set of forward, backward, and correspondence analysis rules.

The overall algorithm consists of two main phases: The first one deals with existing links, while the second one aims at handling nodes that are not referenced by links, yet, and at creating new links.

Each link in the integration graph has an associated state. When a link has been newly created by executing a rule or manually by the user, its state is initially set to consistent. In the first phase, for each link it is checked whether source and target patterns originally referenced by the link are still present in source and target graphs. If some parts of the patterns are missing due to modifications of source and/or target graphs, the state of the link is changed to damaged.

In the next step, it is attempted to repair damaged links. There are different possible repair strategies, most of which require user interaction. Some of these possibilities are explained in Section 3.3.

In the second main phase of the overall algorithm, forward, backward, and correspondence analysis rules as introduced in Sections 3.1 and 4.1 are applied to nodes that are still available, i.e. are not referenced by a link, yet. After all rules have been applied, there may still be some nodes that have to be dealt with manually, due to the lack of appropriate rules. Additionally, links created by executing rules may be modified by the user later on. As already mentioned in Section 2.1, we do not intend to provide a

fully automated transformation. Instead, we explicitly support the combination of manual and automatic steps to perform the transformation. This phase of the algorithm is described in more detail in Sec. 3.2.

3 Graph-Based Concepts

In this section, the underlying graph-based concepts and graph algorithms for integration tools are presented. First, we motivate how our integration approach is related to triple graph grammars (Subsec. 3.1). Then, in Subsec. 3.2 we present in detail the algorithm supporting user interactivity (according to the overall algorithm) and in Subsec. 3.3 we explain how existing relationships are checked for consistency and repaired, if needed.

3.1 Triple Graph Grammars as Conceptual Basis

Triple graph grammars [7] were developed for the high-level specification of graph-based integration tools. The core idea behind triple graph grammars is to specify the relationships between a *source*, a *target*, and a *correspondence graph* by *triple rules*. A triple rule defines a coupling of three rules operating on source, target, and correspondence graph, respectively. By applying triple rules, we may modify coupled graphs synchronously, taking their mutual relationships into account.

Comparing to complex scenarios as described in the previous section, we have a *source document* and a *target document* which may be modeled as graphs, and moreover, the operations on the documents performed by the respective tools may be modeled by graph transformations. The data structure storing links between inter-dependent documents maintained by an integration tool has been called *integration document* which also can be modeled as graph. So, we have the three graphs involved, source, target, and correspondence graph. Integration rules correspond to triple rules.

We have developed prototypes based on the UPGRADE framework [20] which allow construction and modification of source and target graphs as well as simulating runs of the integration tool. Such prototypes are intended only to serve as proof of concept and for the evaluation of integration rules as well as the integration algorithm itself and are referred to as IREEN (*I*ntegration *R*ule *E*valuation *EN*vironment) [21] prototypes. Source and target graph specifications are written and compiled by the PROGRES system [22] and then are embedded into the UPGRADE framework [20]. Triple rules are defined by PROGRES transformations.

An example of a triple rule is given in Figure 5 in PROGRES syntax. The rule refers to the creation of *connections* (appearing in both PFDs and simulation models). In a PFD, a connection is used to relate structural elements such as *devices* and *streams*. In Figure 1, devices are represented as rectangles, streams are shown as directed lines. Connections are not represented explicitly, but they are part of the internal data model. Each device or stream has a set of ports; connections establish relationships between these ports.

The triple rule `ConnectionSynchronous` has a *left-hand side* (shown above the right-hand side) which spans all participating subgraphs: the source graph (representing the flow sheet) on the left, the correspondence graph in the middle, and the target

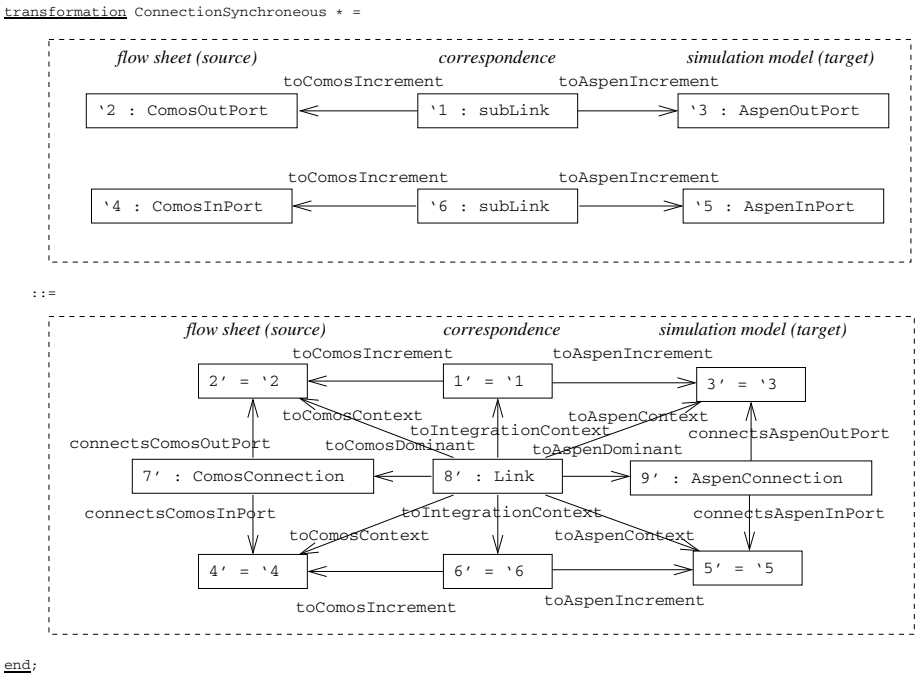


Fig. 5. Triple rule for a connection

graph (for the simulation model) on the right. The left-hand side is composed of port nodes in source and target graph, distinguishing between output ports and input ports. Furthermore, it is required that the port nodes in both graphs correspond to each other. This requirement is expressed by the nodes of type `subLink` in the correspondence graph and their outgoing edges which point to nodes of the source and target graph, respectively. Port correspondences are established by other triple rules which transform the blocks the ports belong to, e.g. streams or devices. Correspondences between source and target patterns are represented by *links* and can be further structured by *sublinks*, e.g. to express port correspondences.

All elements of the left-hand side re-appear on the *right-hand side*. New nodes are created for the connections in source and target graph, respectively, as well as for the link between them in the correspondence graph. The connection nodes are embedded locally by edges to the respective port nodes. For the link node, three types of adjacent edges are distinguished. `toDominant` edges are used to connect the link to exactly one *dominant increment* in the source and target graph, respectively. In general, the source and target pattern related through the triple rule may consist of more than one increment in each participating graph. Then, there are additional edges to *normal increments*[1]. Finally, `toContext` edges point to nodes which are not themselves part of the

[1] The distinction between dominant and normal increments is not vital, but helpful for pragmatic reasons.

transformation but are required as a context condition. These nodes are called *context increments*.

Figure 5 describes a *synchronous graph transformation*. In case of asynchronous modifications, the triple rule shown above is not ready for use. So, we derive *asynchronous rules* from the synchronous rule in the following ways:

- A *forward rule* assumes that the source graph has been extended, and extends the correspondence graph and the target graph accordingly. Thus, the forward rule derived from our sample rule would contain node 7 on the left-hand side.
- Analogously, a *backward rule* is used to describe a transformation in the reverse direction. In our example, node 9 would be part of the left-hand side.
- Finally, a *consistency analysis* rule is used when both documents have been modified in parallel. In our running example, this means that connections have been inserted into both the PFD and the simulation model and a link is created a-posteriori. Thus, the consistency analysis rule would include nodes 7 and 9 on the left-hand side.

Unfortunately, even these rules are not ready for use in an integration tool as described in the previous section. In the case of non-deterministic transformations between inter-dependent documents, e.g. look at the scenario from Section 2.1 where the engineer has to select among multiple alternatives how to transform a reactor, it is crucial that the user is made aware of conflicts between applicable rules. Thus, we have to consider all applicable rules and their mutual conflicts before selecting a rule for execution. To achieve this, we have to give up *atomic rule execution*, i.e., we have to decouple pattern matching from graph transformation.

The approach is that from the set of synchronous triple graph grammar rules for a specific pair of documents we derive a set of PROGRES graph transformations, where graph pattern matching and transformation are handled separately, and combine them with some invariant parts of a PROGRES specification. The resulting overall specification is executable, which allows the evaluation of the integration algorithm and the integration rule set. The algorithm for the execution of integration rules is described in the next section.

3.2 Interactive and Incremental Transformation

Breaking up atomic rule execution into *pattern matching* and *graph transformation* activities is realized in two steps. First, from each synchronous triple graph grammar rule contained in the rule set a forward, a backward, and a correspondence analysis rule are derived as explained in Section 3.1. The integration algorithm for the execution of integration rules consists of rule-specific and generic graph transformation rules. So second, the rule-specific graph transformation rules are derived from the forward, backward, and correspondence analysis rules. Below, one example of the resulting graph transformation rules is described. For a more detailed description, the reader is referred to [23].

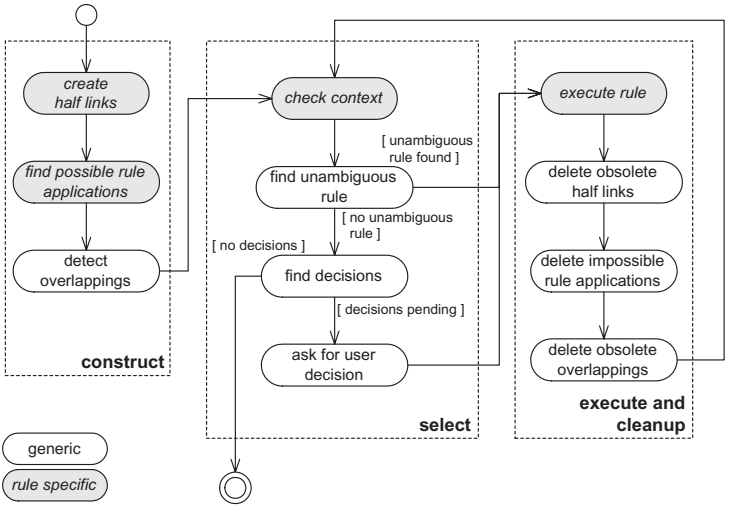


Fig. 6. Simplified integration algorithm (rule execution part)

The integration rule execution algorithm (phase 2 of the overall algorithm, cf. Sec. 2.4) is defined by its overall control structure, depicted as UML activity diagram (with additional annotations) in Figure 6, together with graph transformation rules realizing the single activities.

The set of integration rules used in an integration tool is directly incorporated into the algorithm by realizing some of the activities with graph transformation rules specific for each integration rule contained in the set (marked in gray and labeled in italics in the figure). The remaining graph transformation rules are independent of specific rules.

The algorithm is used to apply forward, backward, and correspondence analysis rules. Here, we briefly present the algorithm focusing on forward rules only. For the sake of brevity, a simplified version of the algorithm is shown and with one exception the steps are explained without referring to concrete rules and their PROGRES transaction counterpart. The complete algorithm with optimization and rules of all directions can be found in [23] and [9].

The algorithm for the execution of integration rules consists of three phases. In the following, these phases are informally described with the help of the abstract example in Figure 7. To keep it small, the example neither relates to specific types of source and target graphs nor to specific rules.

The first phase (construct) is to determine all possible rule applications and conflicts between them and store them in the integration graph. First, for each increment in the source graph that has a type compatible with the dominant increment's type of any rule, a *half link* is created that references this increment. In the example, half links are created for the increments I1 and I3, and named L1 and L2, respectively (c.f. Figure 7 a).

Then, for each half link the possible rule applications are determined. This is done by trying to match the left-hand side of forward rules, starting at the dominant increments to avoid global pattern matching. In the example (Figure 7 b), three possible rule

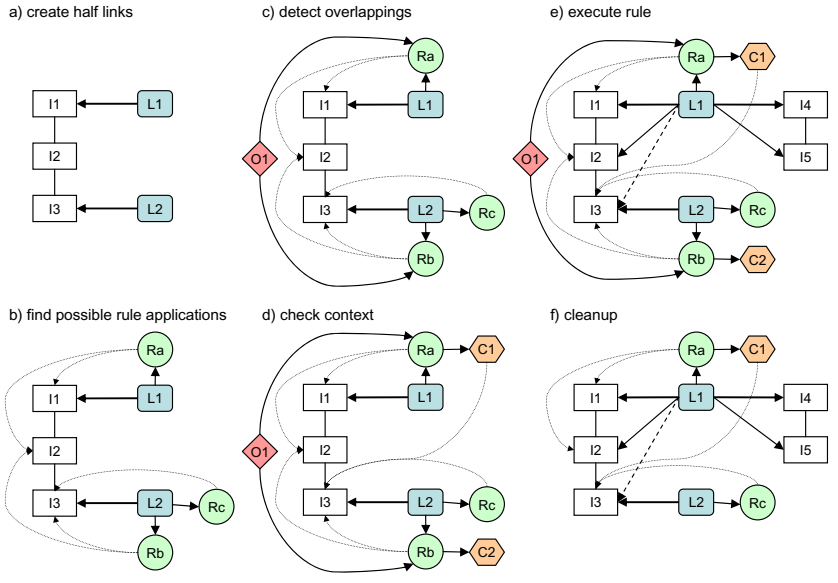


Fig. 7. Simplified example integration

applications were found: Ra at the link L1 would transform the increments I1 and I2; Rb would transform the increments I2 and I3; and Rc would transform increment I3.

Each increment can be referenced by one link only as non-context increment, as increments are "consumed" by integration rules. Thus, a *conflict* occurs if multiple possible rule applications reference the same non-context increment. After applying one of the conflicting rules, they are no longer available for the competing rules. Therefore, in the case of a conflict, the user has to choose one of the conflicting rules in the execute phase.

Whereas conflicts resulting from two possible rules related to one half link are already obvious, those related to overlappings between normal increments of possible rules are explicitly marked in the graph by adding an edge-node-edge construct (e.g., O1 in Figure 7 c).

In the next phase (context check), the context is checked for all possible rule applications and all matches are stored in the graph. Only rules whose contexts have been found are ready to be applied. In the example in Figure 7 d), the context for Ra consisting of increment I3 in the source graph was found (C1). The context for Rb is empty (C2), the context for Rc is still missing.

To illustrate how PROGRES transformations are composed for the rules of the scenario in Section 2.1, Figure 8 shows the transformation for checking the context of the connection rule. The left-hand side contains the half link (`7), the non-context increments (here, only `3), the rule node (`5), and the role nodes (`1). The non-context increments and their roles are needed to embed the context and to prevent unwanted folding between context and non-context increments. For the example rule, the context

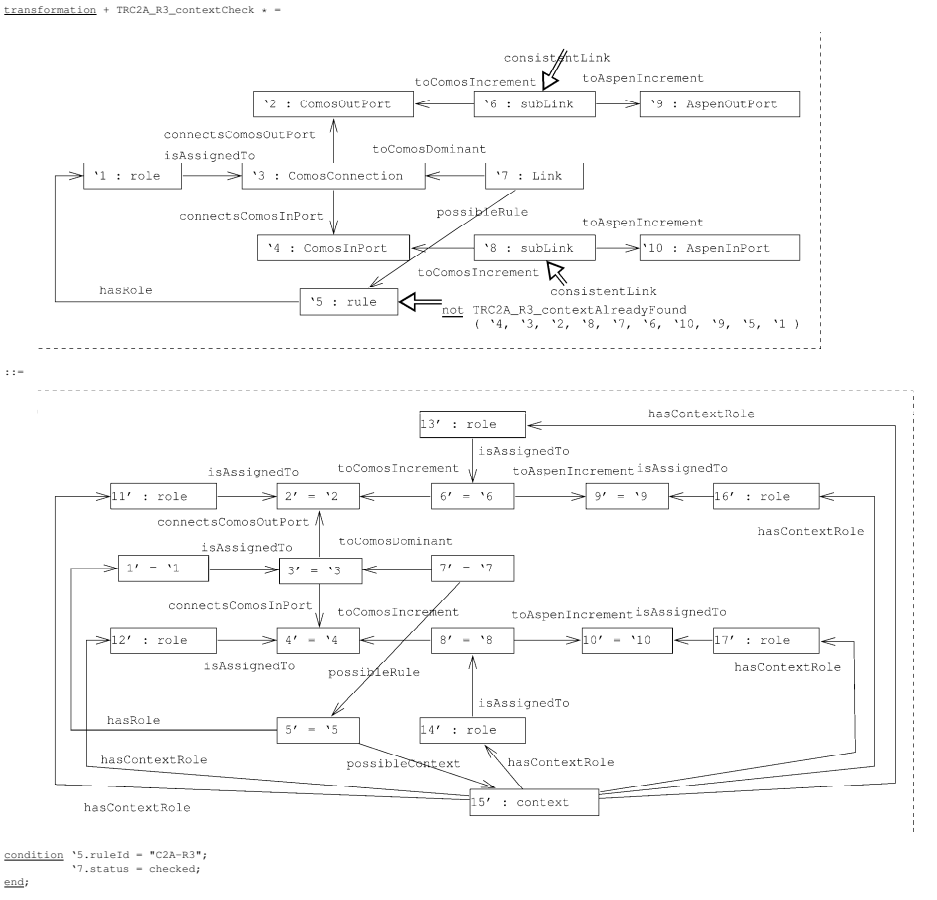


Fig. 8. Check context

consists of the two ports connected in the source document (`2, `4), the related ports in the Aspen document (`9, `10), and the relating sublinks (`6, `8).

On the right-hand side, a new context node is created (`15). It is connected to all nodes belonging to the context by role nodes (11′, 12′, 13′, 14′, 16′, 17′) and appropriate edges. If the matching of the context is ambiguous, multiple context nodes with their roles are created as the production is executed for all matches.

In the last phase (execute), a rule is selected for execution and after its execution the data collected in the construct phase is updated. If any rule application whose context is present is unambiguous, i.e., it is not involved in any conflicts, it is automatically selected for execution. Otherwise, the user is asked to select one rule among the rules with existing context. If there are no executable rules left, the algorithm ends. In the example in Figure 7 d), no rule can be automatically selected for execution. The context

of Rc is not yet available and Ra and Rb as well as Rb and Rc are conflicting. Here, it is assumed that the user selects Ra for execution.

Then, the selected rule is executed. In the example (Figure 7 e), this is the rule corresponding to the rule node Ra. As a result, increments I4 and I5 are created in the target graph, and references to all increments are added to the half link L1. Now, the half link has become a *consistent link*, also called full link. The result in source, target, and integration graph—concerning the link in question and its increments—is the same as if the corresponding forward triple graph grammar rule had been applied in an atomic way.

Afterwards, rules that cannot be applied and links that cannot be made consistent anymore are deleted. In Figure 7 f), Rb is deleted because it depends on the availability of I2 which is now referenced by L1 as a non-context increment. If there were alternative rule applications belonging to L1, they would be removed as well. Last, obsolete overlappings have to be deleted. In the example, O1 is removed because Rb was deleted.

Now, the execution returns to the context check phase, where the context check is repeated. Finally, in our example the rule Rc can be selected automatically for execution because it is no longer involved in any conflicts if we assume that its context has been found.

3.3 Interactive Repair Actions

In this section, we are addressing the first step of the overall integration algorithm (cf. Section 2.4) when modifications on one or both documents have damaged the consistency links between these documents. We want to consider real-life scenarios where the documents' editing tools are decoupled from each other. Thus, we assume not to have any knowledge about the changes on the graphs and have to perform a damage check when the integrator starts, which determines all damages to previously consistent links caused by users' editing activities in the meantime. Damages are all violated conditions of the rule which originally produced the link and all referenced increments, i.e., the graph pattern of the rule does not match, attribute conditions are violated, required dependencies to other links are not valid. We now study how to come to a consistent situation again.

Links are Damaged during Development and Maintenance. In order to explain the possibilities when and how a link may be damaged, we regard a small example based on the example from Figure 1. A part of the underlying internal model of this scenario is shown as a graph in Figure 9. The corresponding user view on source and target graphs is depicted on the left and right,respectively, relating the model to the scenario from Figure 1. In this example, the user had deleted the RPlug reactor (marked with a cross) in the simulation model. In the internal model, the streams Str1 and Str2 to the REQUIL reactor and the HEATER still exist.

The following possibilities exist for a link to be damaged:

1. *Increments* of the source or target graph which take part in a link have been *deleted*, resulting in dangling references of the link. As an example, in Figure 9 the increments TP1, TP2, and R1 have been deleted.

2. Attribute *values* of source or target nodes of a link have been *changed* such that attribute conditions do not hold any more. This would be the case if an attribute of Pl-Rea characterizing the plug flow reactor would be directly changed.
3. *Edges* of the source or target graph being involved in a link have been *deleted*, such that the patterns on both sides are no longer valid.
4. A link of the integration context of the current link has become damaged. In the example, L1 became damaged by the deletion of some increments, thus, all links referring to L1 or one of its increments as context become damaged, too. This is the case for the connection mapping link L2.

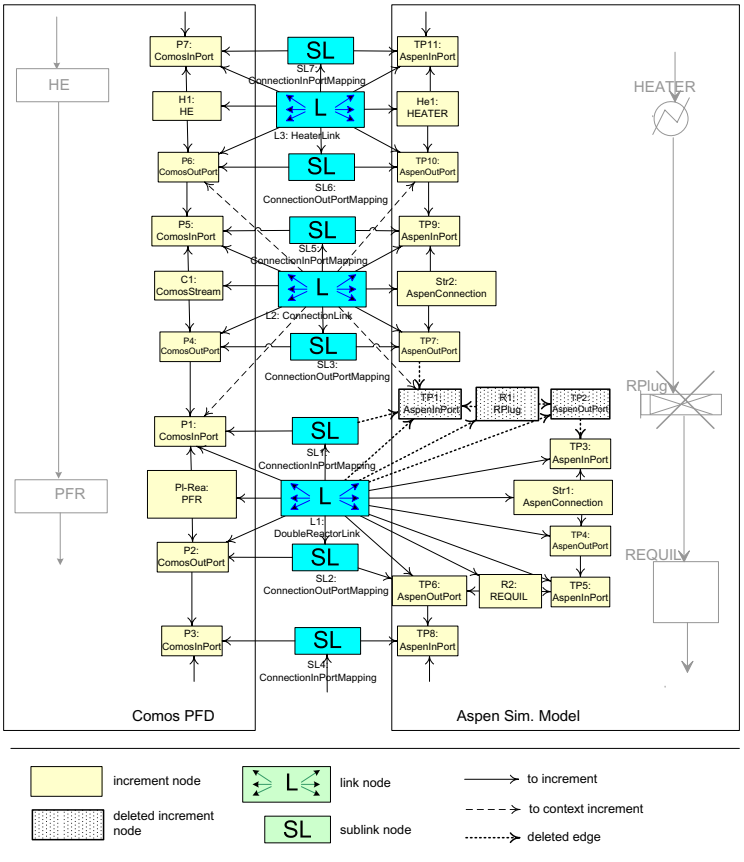


Fig. 9. Reason for a link to be damaged (internal graph view)

Necessity and Types of Repair Actions. We now discuss what has to be done if damaged links have been found. There are three **simple possibilities** we regard first:

1. The most primitive procedure is to *delete the link and all the increments* on both sides, which are involved in the integration situation (in Figure 9 all the nodes and edges of that picture linked to link L1). It is obvious that this is suitable only in

situations where all increments of a link have been deleted by the user in one of the graphs or where one of the dominant increments has been deleted. If some increments of a link in one of the graphs are deleted and others still exist, such as in the example from Figure 9, the deletion could have been part of a restructuring activity, thus, deleting all increments is not the reaction the user expected. Still, it is a valid repair action, but one which should never be executed without prompting the user before.

2. The next simple possibility is to *only delete the damaged link* and to leave the nodes on both sides unchanged and to make them available for other integration rules. Most of the time this also does not lead to the desired behavior of the tool. The modifications resulting in the damaged link were probably done on purpose. The link, although being damaged, may contain valuable information to be used, especially, to determine which parts of the document may be affected by the modification which damaged the link. For example, the link L1 from Figure 9 stores the information that the remaining connection Str1 in the simulation model was produced by the application of the integration rule for mapping a plug flow reactor in the PFD to a reactor cascade in the simulation model. Thus, by deleting the link L1 this connection Str1 would remain in the simulation model uselessly. Furthermore, re-integration may result in at-cross transformation, i.e., a forward and a backward transformation of the remaining increments (in the case of the connection Str1 a corresponding connection in the PFD could be produced).
3. Another option is to restore consistency by *removing the cause for the inconsistency*. For instance, missing increments or edges may be created. This option is desirable only in those cases where the operation causing the damage was carried out accidentally, because it would be undone. For attribute values, the attribute conditions of the synchronous rule can be used to propagate the change.

As the three simple alternatives to (formally) repair inconsistencies presented so far are not very useful from the practical point of view, more specific repair actions [10] based on knowledge about the original rules which created the links have to be considered. They are called *repair actions*, as they bring a damaged link back to a consistent state. The main priority when repairing links is to *conserve the rule* that has been originally *applied* to create the link or to *substitute* it with a *similar* one.

In the following, we discuss why **substitution with a similar rule** is useful and what similarity between rules means. Usually, for the transformation of a pattern of increments from the PFD to a corresponding pattern in the simulation model, multiple alternative rules could be applied. There are rules for the transformation which *differ* corresponding to (1) *multiplicity*, (2) *type*, or (3) swapped elements or (4) the patterns to be created have a subset relation. They are regarded to build up a *rule group*, as any of theses rules may be chosen. Rule of the same rule group are called to be similar.

For example, (1) a plug flow reactor of the PFD may be inserted as a single reactor (rule 1) or a cascade of multiple reactors (rule 2) in the simulation model. This can be applied to the situation in Figure 9: The multiplicity of the reactors has been changed (removal of RPlug). Now, instead of applying any of the general repair strategies listed above, another rule from the rule group is chosen, leading to the situation shown in Figure 10 where the link now refers only to one reactor and the connection Str1 is

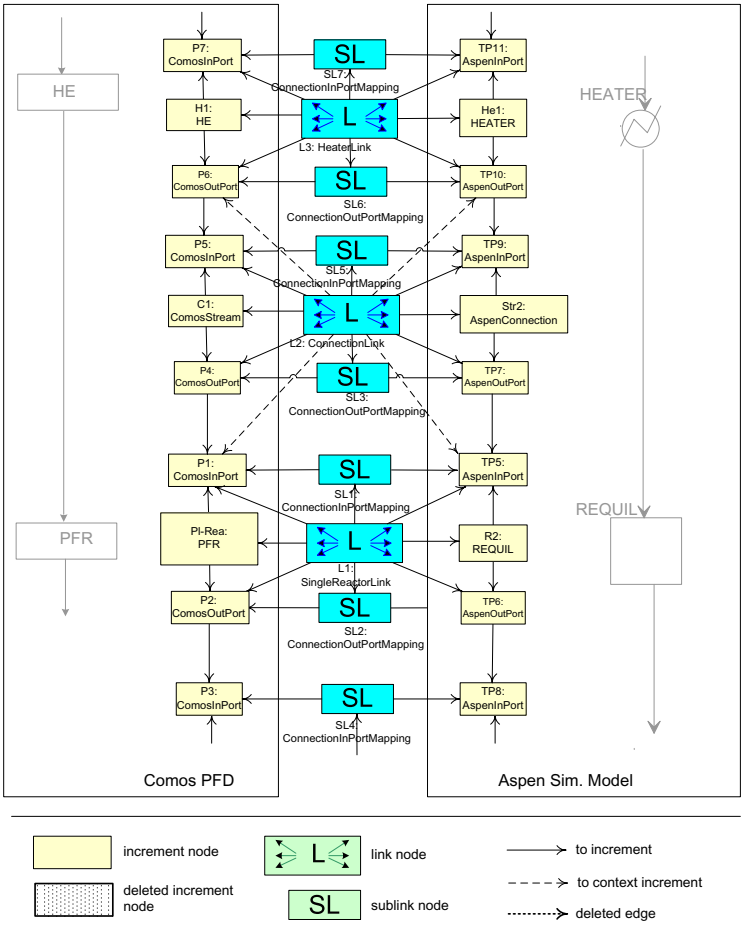


Fig. 10. Damaged link from Fig. 9 now repaired (internal graph view)

removed. The sublink SL1 now refers to TP5 and TP5 is connected to the out port TP7. The repair action consists in changing the situation as if the singleton rule - on the right side - would have been applied.

Another situation (2) also often occurring in practice is that the elements in the PFD or simulation model are changed by their type and the connection structure remains unchanged. Here, the best alternative for repair in case of a damage is to try to find a rule from the rule group with matching type, maybe including the propagation of the type change from one document to the other. Please note that the graph structure did not change at all.

Also, (3) there might be rules for refining a PFD element in a simulation model, which differ only such that different alternative details may be contained in the pattern on the simulation model side or not.

Another case is (4) that nodes of the overall pattern of one rule are a subset of the pattern of another rule. The applicability of a subset rule is likely as a subset rule has less conditions which could be damaged. If for a damaged applied rule there exists a subset rule which is not damaged, then increments which are not represented by the subset rule but by the damaged applied rule are deleted.

Conserving the applied rule means to heal single damages of the rule pattern with the following sub-strategies.

Attribute propagation. Attribute values are changed so that a violated attribute equation holds again.

Attribute restriction repair. Attribute values are changed so that a violated attribute restriction holds again.

Alternative nodes and edges. Missing nodes and edges can be replaced by *alternative nodes and edges* which exist in the respective graphs if they are not used by another link.

Alternative context. Missing nodes and edges playing the role of the *context* of a damaged link can be replaced by *alternative nodes and edges* in the respective graphs if they are not used by another link.

Complete rule pattern. Missing nodes and edges can be recreated, thus, the pattern of the applied rule is completed, resulting in an *undo*-like operation of the user's changes. Note that edges which were deleted due to the deletion of incident nodes are not recreated as well as attribute values of recreated nodes.

NAC repair. A node violating a negative application condition (NAC) [24] condition of the applied rule is deleted.

We now summarize all repair types in the following list and describe their strategy in short.

Delete rule. All remaining increments participating in the same link are deleted as well. Also the link and all its sublinks are deleted (corresponds to simple possibility 1 above).

New integration. In this naive approach the link is deleted and the increments are thus *freed* for another transformation, but as discussed above, acts as *last resort* (corresponds to possibility 2 above).

Undo damaging changes. This useful repair type cannot be performed by the integration tool for the above mentioned reasons. It could be realized by support of the editing tools if the tools provide incremental and selective undos (corresponds to simple possibility 3 above).

Conserve rule. Single damages of the rule pattern are healed with sub-strategies. They are selected at runtime dependent of the damage types and number and combined together (see below).

Alternative rule. It can also be possible to apply a similar rule with little adaptations to the graph; in case that some preconditions for applicability of the rule are not fulfilled, their validity can be enforced by little changes on the graph.

New correspondence. Like alternative rule repair, this repair action searches for another possible rule which can be applied. But instead of taking the whole pattern

into account, it just searches for source and target dominant increments, as a minimal requirement for an alternative rule application if they still exist (slight variant of possibility 2 above).

Define new rule. This repair type is a trivial solution to handle damage repair as for the new situation a rule is defined (induced) and attached to the link.

It is important to distinguish the role of a missing node by context or non-context when searching for an alternative node because a repair action is allowed to change only nodes and edges that were originally created by the applied rule. That does not hold for context increments. Therefore, a repair action does never cause new inconsistencies for links referenced as context. But of course, other links having a repaired link in their contexts could be damaged.

As mentioned, the repair types listed above are strategies from which concrete instances (repair actions) are derived at runtime dependent on the damaged rule and the specific damages triggered by the user's changes. Repair types/actions are divided into those which adapt source and target graphs and those which only adapt references within the integration document assuming that the user established a consistent state by himself. The first ones are called *changing* and the latter ones are called *non-changing*. Therefore, repair type Delete rule[2] counts to the changing, whereas the others do not change the documents.

If a Conserve rule repair action is a changing or non-changing repair action, depends on the damages of the current rule to be conserved. If only non-context increments and edges are missing, i.e. no attribute or NAC conditions are violated, and they can be substituted with Alternative nodes and edges, it is a non-changing repair action because it only makes changes in the integration document. If an Alternative context has to be found this can result in a changing and non-changing repair action. The non-changing version searches for the required nodes and edges and adapt the references in the integration document to source and target graph only. The changing version reassigns edges from the context to source and target graph.

A repair action of type Alternative rule also can be changing or non-changing. The changing version creates required nodes and edges for the application of a different rule and may delete the rest of nodes and edges of the former applied rule which are not matched by the alternative rule.

Generating Repair Actions. As for the incremental integration algorithm from Section 3.2 user actions in the source or target document are not recorded due to independent use of corresponding tools. Before executing repair actions, as a first substep of the first phase of the integration algorithm, all existing links have to be checked for damages. This is possible as the links are kept separately in the integration document and contain all information needed for the check, like the rule having been applied, the necessary context, the dependency of a link w.r.t. other links, and alike. If a link is damaged, all other links referencing it as context have to be set to damaged as well.

The derivation of forward, backward, and correspondence analysis rules at specification time as described in Section 3.1 is very pragmatic. However, it is not possible to

[2] And all derived actions.

derive all repair actions, which are candidates for repairing inconsistent integration situations. For a triple rule which contains n nodes and e edges there exist 2^{n+e} graph patterns describing all possible inconsistent situations, i.e., each combination of a deleted node or edge. Even more, further nodes and edges might have been inserted, values might be inconsistent, for any inconsistent situation there might be different repair options etc. So, the number of possible transformations of repair actions for inconsistent situations further explodes. All of these transformations would have to be used to be checked for applicability.

Therefore, we chose to construct specific repair actions (i.e. graph transformations) for a damaged link after the damage check at runtime. The repair actions represent alternatives to make the link consistent again. This means, the graph part of a damaged link which still conforms to the corresponding integration rule (*remaining graph*) is the basis of the left hand side (LHS) of all repair actions generated for this link. From the remaining graph a corresponding graph pattern for the LHS of all the alternative repair actions is built and its pattern nodes are preset to point to the actual nodes of the remaining graph before pattern matching.

Depending on the repair type of a repair action, further pattern nodes and edges must be added to the LHS without already pointing to actual nodes. The approach specifies LHS and RHS of the graph transformations for each repair type (intensionally) by set-oriented descriptions [10]. The exact structure of a graph transformation is computed when repair actions are constructed. It depends on the damaged rule and the certain damage. I.e., the intensional description does not know the rule pattern and takes all missing nodes as variables, the specific computed graph transformation retrieves the concrete rule pattern and replaces the variable nodes by concrete ones.

If the applicability of an alternative rule is searched then the set difference of the alternative rule specifying pattern and the remaining graph pattern is included in the LHS. The RHS is basically equal to the LHS but contains additional pattern nodes and edges, i.e., those which must exist according to the alternative rule but are not existent, and some are no longer present in the RHS, i.e., those which are not used by the alternative rule. Also, attribute values are changed if required.

The strategy to conserve the applied rule also leads to multiple repair actions: a non-changing repair action to conserve the originally applied rule without doing any changes on the graphs or multiple changing versions of that repair action (i.e., missing nodes and edges can be created here with the create rule pattern strategy). Since there can be multiple reasons for a link to become damaged and, as described above, for each of these reasons some sub-strategies are defined, the repair actions to conserve the applied rule are constructed stepwise; in each step, each of the repair actions are extended to *heal* a certain actually existing damage. Extending a repair action consisting of a LHS and RHS is done by gluing the LHS and RHS of a respective *component* graph transformation which represents the change for healing a certain damage. For each sub-strategy, component graph transformations are also specified (intensionally) by set-oriented descriptions.

There are repair actions created for deleting all increments of a damaged link (delete rule) or allowing a new integration by deleting the damaged link and all of its sublinks. The definition of a new rule is also a repair action which - when applied - establishes

a state that the integration tool would recognize as consistent. When executed, the integration tool additionally derives a new integration rule, but this is done outside of the repair action. Also, for these alternatives set descriptions are defined, the construction consists only of a single step.

To give an example, the deletion of the RPlug R1 from the simulation model from Figure 9 led to two damaged links, L1 and L2. For L1, increments R1, TP1, and TP2 are missing, for L2 the context increment TP1 is missing. The LHS of all the repair actions generated consist of all the nodes L1 and L2 still refer to and all existing edges between them. For L2 these are the normal and dominant increments C1, P4, and P5 (source graph), Str2, TP7, and TP9 (target graph) and the context increments P1 and P6 (source graph) and TP10 (target graph). To solve the damage of L2 the repair action is extended according to the sub-strategy Alternative context. To find an alternative context node, this node must appear in the LHS. The node is determined by performing a set-subtraction of the current LHS and the LHS of the original triple rule which returns a node of the type of the missing context increment (here AspenInPort) which is connected to the context sublink SL1. The RHS of the repair action consists of the same pattern as the LHS and is extended by the `toContext` edge referring from L2 to this *alternative context* node and the edge from TP7 to that node. Adding these edges is also the result of performing a pattern difference of the current RHS of the repair action and the RHS of the triple rule.

To solve the damage of L1 one possibility is to apply the *single reactor rule* which is a subset of the originally applied *double reactor rule*. Thus, by performing the pattern differences of the current LHS and RHS of the repair action and the LHS and RHS of the triple rule, the AspenConnection Str1 and its ports TP3 and TP4 are removed from the RHS and an edge from the sublink SL1 to the in port TP5 of the reactor is added to the RHS.

The integration tool could now remove the damages of the scenario by first applying the repair action for reparing the link L1 which produces a new in port node of the sublink SL1, so the repair action for L2 can be applied afterwards, but this is not known at the time when the repair actions are generated.

In the example, both repair actions are constructed in one step. If there are more damages, e.g. violated attribute conditions or further nodes missing, additional nodes and attribute transfers would be integrated into the repair action as the result of the difference computation with the original triple rule. In the example, if more nodes would have been deleted by the user in source or target graph which were also part of the subset Single Reactor rule, the result of the pattern difference would lead (a) in the non-changing version (to include already existing nodes as alternatives) to an extension of the LHS *and* an extension of the RHS and (b) in the changing version (to create the missing nodes) to an extension of the RHS only.

Execution. In general, it cannot be determined automatically which alternative for repairing damaged links is appropriate. Because of that, user interaction is necessary here as well resulting in breaking up atomic transformation execution, i.e., decoupling pattern matching from graph transformation, analogously to the approach of integration rule execution (cf. Section 3.2). The integration tool finds all possible repair actions,

then, all possible applications of normal integration rules are determined (construct phase), and both kinds of possible executions are presented to the user who chooses one, either being a repair action or a transformation rule (select phase). The selected transformation is then executed, other competitive rule and repair actions are deleted (execute and cleanup phase).

Optimizations. Generating repair actions at runtime is suitable for the above mentioned reasons – one argument against deriving repair actions at compilation time was combinatorial explosion of repair actions – but this approach also implies performance problems when at runtime a set of repair actions is generated, most of them not being applicable as required, e.g., required nodes and edges of the LHS are not present in the graph. Thus, we have optimized the generation by distributing priorities among the repair types and construct step-wise repair actions according to the priorities and subsequently test their applicability until some are applicable. The priorities can be configured by the user. In Section 4.3 the time complexity of the implementation and in Section 4.4 the user interface for configuring the priorities are discussed in more detail.

4 Transfer to Practice: User Interfaces and Evaluation

Although the concepts presented above suffice for construction of integrators, they still lack usability for an industrial setting: there is the need to simplify user interaction by developing graphical user interfaces for configuration, simplifying the rule specification language, and supporting the rule generation process. All these needs are addressed in the transfer phase [25,17] of the IMPROVE project [8] resulting in the extension of integrator facilities and in the development of further management tools, which are discussed in the following.

This section is devoted to give an overview of the developed user interfaces and how possible repair actions are presented to the user. Furthermore, it summarizes our experiences about performance and quality of the offered repair actions.

4.1 Rule Modeling Support

Document-specific Rule Editors for more Acceptance. A basic implementation of a rule editor based on Microsoft Visio drawings which allows rule modeling with UML object diagrams [18,19] is provided. The rule editor can be easily extended: instead of showing plain UML syntax, the documents' increments are shown in a syntax similar to those of the tools to be integrated. For example, for modeling cutouts of PFD documents which are edited with COMOS, icons representing devices of PFDs are placed within the object of the diagram. This is achieved by adapting the *type change event* of the rule editor. When the rule modeler changes the type of an object of the rule, COMOS is accessed and the icon for the new type is retrieved. By adapting the *paste event* it is easy to achieve a tight integration of the document editing tools with the rule editor allowing to select and copy objects of a document editing tool and to paste these objects into the rule editor. This adaptation was implemented for the COMOS tool, too.

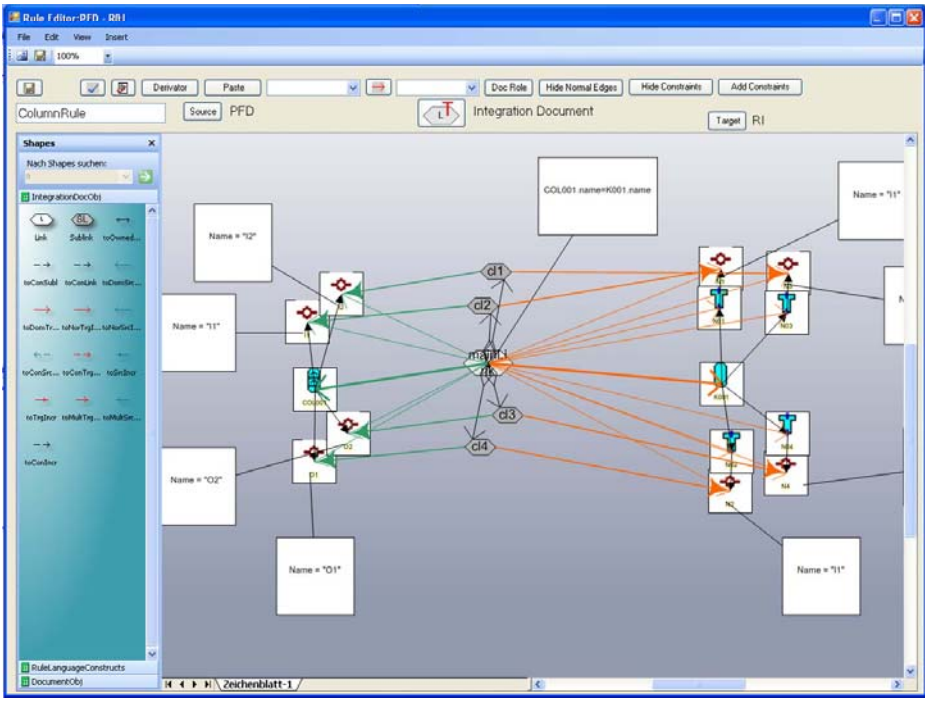


Fig. 11. Rule editor with PFD and P&ID specific looks for devices

As mentioned in Section 2.2, we also have built integrators for other pairs of documents within the transfer project, such as for integrating PFDs and piping and instrumenation diagrams (P&ID), which are both edited with COMOS. A P&ID is more detailed than the PFD as it is a blueprint of the chemical plant. How a rule for this integration scenario looks like is presented in Figure 11. It specifies how a column and all of its ports of a PFD corresponds to a specific column and its ports in a P&ID. Usually, columns in P&IDs have more ports as in PFDs (not in this rule) and ports are connected to the columns via nozzles. All these objects contain the icons from COMOS such that a rule modeler who is familiar with COMOS recognizes the elements immediately.

There are many other supporting features coming along with the editor [10], e.g. ensuring the use of unique names of the objects within a rule or the possibility of executing self-written scripts on the rules which is very useful when the rule modeler wants to adapt his rule mechanically. But the features which got the most attention of the industry partner to encapsulate the complex concepts behind rule modeling were (i) the rule checker which marks errors in a rule, (ii) the event handlers mentioned above which allow integration with the tools that the user is familiar with, and (iii) usage of increment-specific attributes. The latter feature is simple but extremely helpful in practice. Apart from the predefined increment attributes `Name` and `Type`, the rule modeler can use arbitrary attribute names within attribute restrictions and equations. The document wrapper interprets these attributes at runtime. For example, attributes of COMOS

objects, the so-called *specifications*, are identified via strings. As an example, the working temperature of a pump is identified by PI020.PIA006 which means it is allowed to use this string in an attribute equation of a rule.

In chemical engineering, the increments of a PFD have many attributes. Furthermore, most values are transferred to the next lower level P&ID. So, rule modeling for two such diagrams means also defining explicitly all the existing attribute relationships which is very laborious. COMOS already defined these attribute relationships for two corresponding devices, i.e, one device in a PFD and the corresponding device in a P&ID, in so-called *mapping tables*. To profit from these mapping tables, we defined a pseudo-attribute called `Attributes` for increments within rules, so many rules contain the equation `PFDIncr.Attributes = PIDIncr.Attributes`. When a forward or backward transformation of such a rule is executed, the COMOS wrapper searches for the mapping table of the respective two increments and executes a COMOS-specific command (here `CalculateLinkedSpecifications`). In general, it is the wrapper's task to determine if and how the pseudo-attribute `Attributes` is represented in the underlying document.

Semi-Automatic Support in Finding Correspondences between Documents and Generalization as Integration Rules. Building integration rules from scratch is tedious. Rule modeling is supported in multiple ways. Object structures from concrete documents for which a rule should exist can be copied and pasted into the rule definition in the rule editor. This assumes that the rule modeler already has identified corresponding object structures between existing documents from preceding projects. There is now tool support for finding corresponding object structures by applying a correspondence analysis with specific integration rules (explained below). From correspondences found, concrete integration rules can be generated automatically [26].

When starting with rule modeling for a new domain or a new pair of document types A and B, the type hierarchies (document models) of A and B have to be inspected and analyzed first to determine which structures and types of A and B may correspond to each other. Such structure and type correspondences are the basis for developing rules which map concrete objects and object structures of that type correspondences onto each other. To support the correspondence analysis on type level, the integrator applies a correspondence analysis based on specific integration rules explicitly designed for type hierarchy documents. From correspondences found on the type level, concrete integration rules are derived for the instance level. Of course, these rules are simple and in most cases have to be extended by the rule modeler.

Both correspondence analyses need further rule language constructs. For the correspondence analysis on the document level, graph patterns were needed which describe the coarse structure of corresponding patterns in source and target documents. These have to be flexible, as the concrete structure is not known at specification time and is going to be derived from the documents. Thus, we introduced graph patterns *Loop* [27] and *Choice* which leave open the number of SCoRe graphs and the choice of a specific alternative used in the graph [26]. For the correspondence analysis on the type level there is the problem that types and attributes which in this case are the increments to map onto each other cannot be distinguished only by structural criteria. Thus, we

added similarity comparing functions using string distance measures and the synonym database WordNet [28] to map names of types and attributes.

The process of correspondence analysis is further optimized to reduce the number of possible matches of integration rules containing the new language constructs. Now, the structure of the documents is included. The analysis starts at a user-defined increment in the source as well as at the target document which the rule modeler recognizes as corresponding, e.g., two main devices in PFD and simulation model or the root types in type hierarchies. In both documents possible rule applications are searched locally around the specified increment. A rule which is applicable in source and target document is a valid option and all options are presented to the user who decides which rule to apply. Then the algorithm traverses edges connected to the increments matched before independently in source and target document and searches again for possible rule applications. If correspondences are found in the documents, *concrete* rules are derived.

Managing Rule Sets and Relationships between Rules. Furthermore, functionality for managing *rule modules* is provided to support rule re-use; there often are redundant parts modeled in multiple rules which had to be maintained. Parts of other rules are used in a rule but with different semantics. We have specified the semantics of three relationship kinds between rules. When a rule uses another rule, the relationship is managed. Becoming explicit, the relationships can be involved within rule execution which is thus improved.

Explicit *contained-in* relationships of rules help to find smaller or bigger rules in rule execution; furthermore, *context* relationships between rules exist if the context of one rule is equal to a part of another rule. Based on these relationships, during execution a dependency analysis is performed which determines dependencies between applicable rules and rules depending on those rules as they expect nodes and edges as context. If rules are conflicting, knowledge about which rules get applicable and which do not supports the user making his decision for or against a rule. Additionally, after one rule is executed there are now possibilities to trigger other rules and to transfer node ids to the triggered rules so that they are applied only locally in the graph in the neighborhood of the triggering rule application point. The rules to trigger are specified within the definition of the triggering rule, thus, producing new *trigger* relationships between triggering and triggered rule.

4.2 User Interface for Integration

In Figure 12, the user interface of the integrator is shown; it is designed in close cooperation with our industrial partner Comos Industry Solutions [29]. It allows on the one hand the configuration of the tool, i.e., selection of documents, a rule set, and integration direction (see the lower right corner). On the other hand, it realizes a user friendly view on the integration state by showing the contents of the documents in three tree views. Each increment of source and target document is a node in the left and the right tree view, respectively. A significant symbol indicates the type of an increment. The middle tree view contains the most important elements from the integration document, i.e., the links.

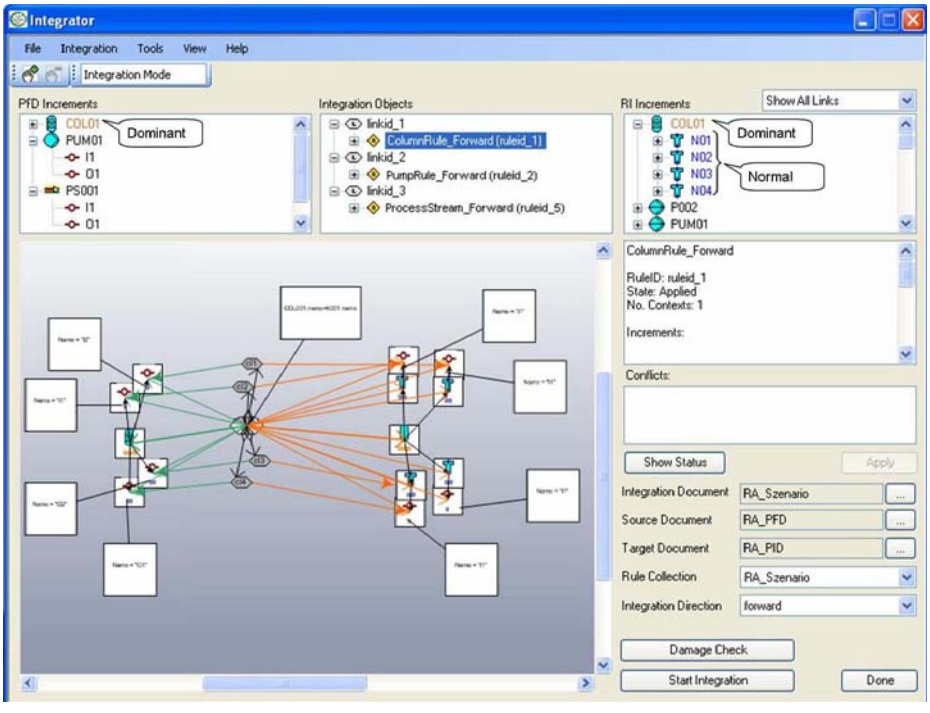


Fig. 12. Screenshot of the integrator user interface after initial forward transformation

If there are rules competing to each other, multiple nodes representing these rules are shown as children nodes. By selecting a rule and pushing the button Apply, the user selects this rule for application. To allow the user to better understand which increments are matched by a potential rule application, the respective increments are colored in the left and right tree view. The rule pattern is displayed in the lower left corner *rule view* where the names of the rule objects are replaced by the names of the respective increments colored equally to the tree views. Thus, the user knows exactly which increments are part of a future transformation.

Figure 12 shows a consistent state after forward transformations have been performed. There are three links created in the integration document and multiple increments in the target document. For example, the column COL01 is transformed by the ColumnRule into a ground column COL01 (dominant increment) which has four nozzles N01 to N04 (normal increments). The according rule is depicted in Figure 11.

The column rule specifies that corresponding columns in source and target document must have equal names. In case, the user changes a name, e.g., the ground column name to K001, the integrator determines that the rule application is damaged and creates repair actions shown as children nodes of the damaged rule as shown in Figure 13. In the figure, there are four repair actions generated: two attribute repairs for transferring the names in either the one or the other direction, a delete repair for removing the whole structure, and an undo of the last change.

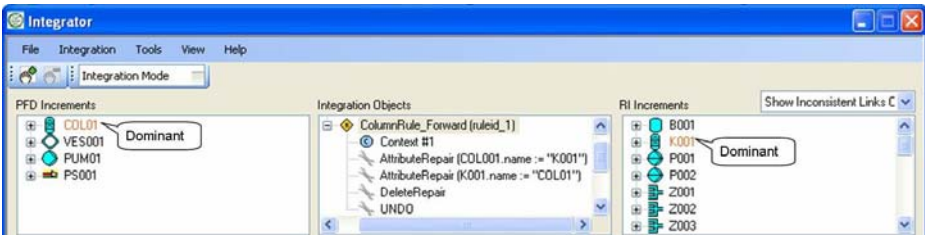
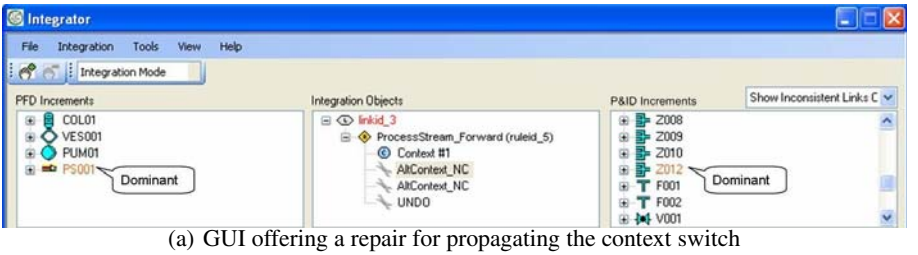
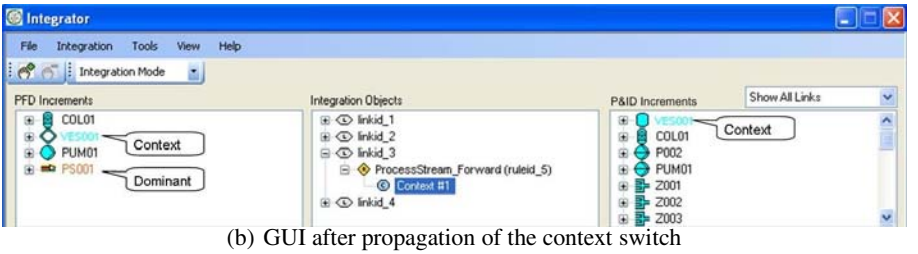


Fig. 13. GUI offering multiple repairs for damaged column rule



(a) GUI offering a repair for propagating the context switch



(b) GUI after propagation of the context switch

Fig. 14. GUI before and after alternative context repair is executed

Figure 14(a) shows a situation where the user deleted the connection of the process stream of the PFD and created one to a vessel, thus, the context of the process stream was switched. The integrator finds two possibilities for repairing the damage as there are two possible contexts: (i) the newly created context consisting of the two vessels from both documents and (ii) the former context consisting of the two columns from both documents as the pipe from the P&ID is still connected to the ground column. The latter is equivalent to an undo repair which is also offered. In this example, the user selects the first repair action. The result is shown in Figure 14(b) with emphasized context nodes, namely the two vessels.

Also, this view was accepted by the industry partner. As with the rule editor, integration with the editing tools is achieved by adaptation of event handlers. From the COMOS tool, PFDs and P&IDs can be dragged and dropped into the left and right tree views allowing to configure the integration tool with a concrete pair of documents. The icons in the tree views are retrieved also from COMOS, when the documents are loaded, which supports the recognition of the elements. When a repair action or integration rule

is selected in the integration document tree view, applied or only as a proposition, the match of increments of the underlying *rule pattern* is presented to the user within the rule view where the name of the rule object is replaced with the concrete name of the matched increment. It is also possible to navigate from an increment representation of the tree views to the respective tool's view. This supports the user a lot in understanding which increments are affected by an integration rule or repair action in the documents.

4.3 Time Complexity and Optimizations

In general, the most critical part of the application of transformation rules (and repair actions) are the pattern matching steps of the algorithm. In the industrial prototypes, all graph transformation operations are performed by a graph transformation component. This component is kept light-weight as only very simple graph transformations have to be executed. All pattern matching is done starting from dominant increments, in most cases only locally traversing the graph avoiding global pattern matching. Thus, the—in theory—high complexity of pattern matching does not affect the integrators' performance.

Generating repair actions for a violated rule has a time complexity polynomial in the number of objects occurring in that rule. In our case studies, the rules considered are assumed to have average size. However, the generation process proved bad runtime behavior due to many repair transformations generated which could not be applied and, thus, were not offered. Therefore, we optimized the process by introducing *phases* where only repair actions for a set of predefined repair types are generated and tested. Only if no valid repair action could be found in one phase, repair actions of another set of repair types are tested in the next phase. This has proved acceptable runtime behavior. The sets of repair types and the order of their execution can be configured by the user. Additionally, a set of repair types can be specified which should always be tested.

4.4 Configuration of Integration Tools

For managing documents and their dependencies as well as rule sets there exist further tools. In Figure 15, the editor for managing documents, i.e. their source (if stored as file in the file system or object in a tool's database) and their connection for integration purposes to other documents, is shown. Documents are represented as document icons and a document relationship is expressed by arrows between two documents. An arrow between two documents contains the source of the integration document, which is the file name in case of a file in the file system or an object id in case of an object in another database.

Documents as well as arrows are attributed with document ids. The document managing tool is integrated with the Integrator GUI such that if a pair of documents is selected for integration in the Integrator GUI, the corresponding integration document and the underlying rule set is activated instantly.

There are several more configuration tools such as the rule collection managing tool which allows for creating rule sets for specific integration situations. A rule can be used in more than one rule set. The idea is that for integrating the same pair of documents different rule sets could be used dependent on the phase of the development process. It is also possible that for pairs of documents of specific document types, e.g., a PFD

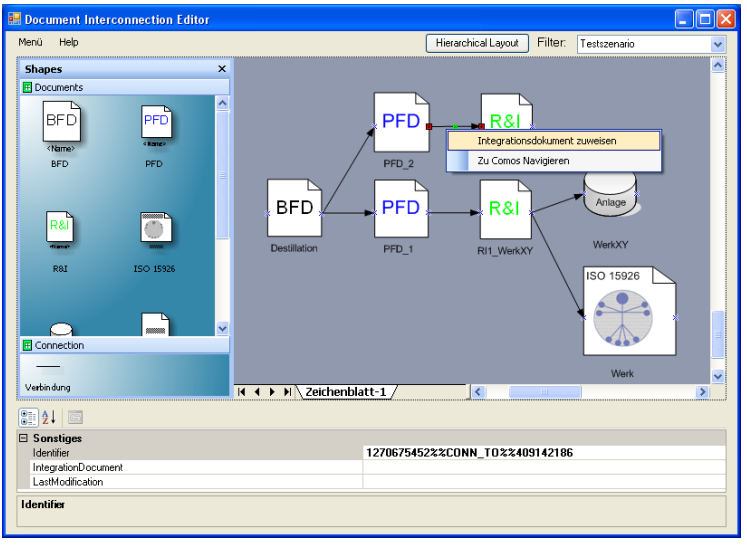


Fig. 15. Document relationship management

and a P&ID, different rule sets could be used. This is the case for example, when the user knows that in one scenario only a small set of devices is used and/or he wants to apply only a limited set of transformation. So he is able to reduce the rules used in an integration scenario.

For the integration itself the user can configure, how integration is performed, e.g., if repair actions are applied and which repair actions are priorized. In Figure 16 the GUI for configuring the phases for the generation of repair actions is depicted. First, the repair actions of the repair types within the first phase are generated and then tested for applicability, second those of the second phase and so on.

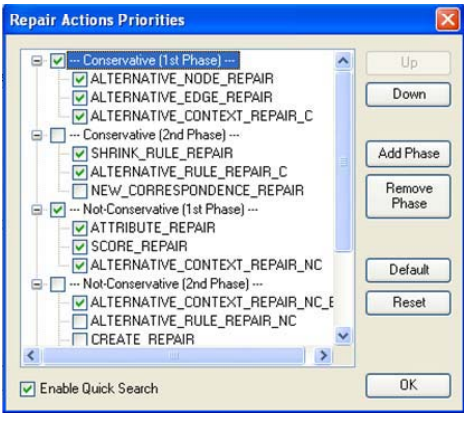


Fig. 16. Configuration GUI of repair types properties and phases

4.5 Evaluation

The whole integrator approach has been closely evaluated together with COMOS industry solutions. The discussion about user interface design and supporting functionality was mostly held with the software tool builders of COMOS. But also potential users have been involved by presenting the approach and prototypes and by collecting and exploiting their feedback. Based on this it can be said that in any case an improvement in usability of the overall integration approach could be achieved. Still, the integrator user interfaces needs to be further optimized to be less complex and even more user friendly for the tools to be ready for the market.

So far, the main contribution of the transfer project lies in the knowledge transfer. Comos Industry Solutions, the software tool builder, was enabled to use the principles of the structure of integration tools to satisfy the demand of their clients for tool support for document integration by including simplified integrator functionality directly into their tools. For the future, it is likely that even more integrator functionality will be included into their tool, probably specific for the existent document types.

Based on this experience, we would recommend that domain-specific adaptations should be done together with domain experts. Furthermore, the user interface for the integration needs to be as easy as possible. For example, the interface should be directly integrated in the document editing tools as plugins, offering easy commands such as start, stop, resume, undo, redo.

5 Related Work

Our approach to incremental integration for development processes is based on the triple graph grammar approach introduced by Schürr [7] and early work at our department in the area of software engineering [6] during the IPSEN project [30]. We have adapted the results to the domain of chemical engineering [31] and extended the original approach: now, we are dealing with the problem of *a-posteriori* integration, the rule definition formalism was modified [18] and the rule execution algorithm was further elaborated to support conflict detection and repair actions (see Sections 3.2 and 3.3).

Related areas of interest in computer science are (in-) *consistency checking* [32] and *model transformation*. Consistency checkers apply rules to detect inconsistencies between models which then can be resolved manually or by inconsistency repair rules. Model transformation deals with consistent translations between heterogeneous models. In the following a few projects of both areas are presented which are using graph transformations. Our approach contains aspects of both areas but is more closely related to model transformation.

In [33], a consistency management approach for different view points [34] of development processes is presented. The formalism of distributed graph transformations [35] is used to model view points and their interrelations, especially consistency checks and repair actions. To the best of our knowledge, this approach works incrementally but does not support detection of conflicting rules and user interaction.

The consistency management [36] approach of Fujaba supports inter-model consistency checks by a plug-in. The approach is based on triple graph grammars [7] as well. Comparable to our approach, different graph transformations are derived from each triple rule. Here in contrast, change propagation is performed immediately. User interaction is restricted to choosing the repair action for a detected inconsistency. Conflict detection between different inconsistency checking rules is supported only w.r.t. preventing endless loops if repair actions create new inconsistencies.

Model transformation recently has gained increasing importance because of the model driven approaches for software development like the model driven architecture (MDA) [1]. In [37] and [38] some approaches are compared and requirements are proposed.

The PLCTools prototype [39] allows the translation between different specification formalisms for programmable controllers. The translation is inspired by the triple graph grammar approach [7] but is restricted to 1:n mappings. The rule base is conflict free so there is no need for conflict detection and user interaction. It can be extended by user defined rules which are restricted to be unambiguous 1:n mappings. Incrementality is not supported.

In the AToM project [40], modeling tools are generated from descriptions of their meta models. Transformations between different formalisms can be defined using graph grammars. The transformations do not work incrementally but support user interaction. Unlike in our approach, the control of the transformation is contained in the user-defined graph grammars.

Re-establishing consistency by transformations is mostly performed in other work in two ways. Either there are only propagations of deletion or attribute changes or deletion of links supported as e.g., in [41,42,43]. These are also provided by our work. Or the reaction on specific inconsistent situations is specified in advance by the rule modeler explicitly as e.g., in [44,45,46,47,48,33]. Manual specification of repair actions is very tedious and would not cover all cases of inconsistencies.

In [42] also the automatic completion of missing increments for applying a rule is supported but only for transformation purposes and not for repairing existing correspondence relations. Another approach similar to ours which constructs repair actions dynamically is presented in [49,50,51]. Consistency rules are specified by first-order formulas. If a term is not fulfilled, model transformations are generated such that the term would be fulfilled after execution. These contain deletion of model elements or changes on model elements, which cause that the term is not fulfilled. The user selects one of the generated transformations. Creation of new model elements is not supported and furthermore the generation of transformations bases on change logs of the actions of the user which cannot be assumed for a-posteriori integration.

Transformations between documents are urgently needed (not only) in chemical engineering. They have to be incremental, interactive and bidirectional. Additionally, transformation rules are most likely ambiguous. There are a lot of transformation approaches and consistency checkers with repair actions that can be used for transformation as well, but none of them fulfills all of these requirements. Especially, the detection of conflicts between ambiguous rules is not supported. We address these requirements with the integration algorithm described in this contribution.

6 Conclusion

We presented the integration tool approach having been developed over many years at the Department of Computer Science 3 chaired by Prof. M. Nagl. The approach was recently evaluated in an industrial cooperation with the German software company Comos Industry Solutions with different prototypes for the integration of PFDs, process and instrumentation diagrams, and simulation models, applied during design processes in chemical engineering. Experiments with the prototypes have shown that our approach considerably leverages the task of keeping dependent documents consistent to each other. Nevertheless, there is still the need for a lot of user interaction. Besides choosing among different possible rules, contradictory changes that have been made to the documents in parallel have to be resolved manually. For this and other aspects, the usability of the approach heavily depends on available GUIs and tool support, e.g. for rule definition. For further reading on this topic, the reader is referred to [10].

Acknowledgments. This work was in part funded by the CRC 476/TC 61 of the Deutsche Forschungsgemeinschaft (DFG). Furthermore, the authors gratefully acknowledge the fruitful cooperation with Comos Industry Solutions.

References

1. OMG Architecture Board ORMSC: Model driven architecture (MDA) (2001)
2. Lewerentz, C.: Interaktives Entwerfen großer Programmsysteme: Konzepte und Werkzeuge. Springer, Heidelberg (1988); Doktorarbeit, RWTH Aachen University
3. Engels, G., Lewerentz, C., Nagl, M., Schäfer, W., Schürr, A.: Building integrated software development environments - part 1: Tool specification. ACM Transactions on Software Engineering and Methodology 2(1), 135–167 (1992)
4. Westfechtel, B.: Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklunsumgebung. Informatik-Fachberichte, vol. 280. Springer, Heidelberg (1991); Doktorarbeit, RWTH Aachen University
5. Janning, T.: Requirements Engineering und Programmieren im Großen: Integration von Sprachen und Werkzeugen. DUV (1992); Doktorarbeit, RWTH Aachen University
6. Lefering, M.: Integrationswerkzeuge in einer Softwareentwicklungsumgebung. Berichte aus der Informatik. Shaker Verlag, Aachen (1995); Doktorarbeit, RWTH Aachen University
7. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
8. Nagl, M., Marquardt, W. (eds.): Collaborative and Distributed Chemical Engineering Design Processes: Better Understanding and Substantial Support Results of the CRC IMRPOVE. LNCS, vol. 4970. Springer, Heidelberg (2008)
9. Becker, S.M.: Integratoren zur Konsistenzsicherung von Dokumenten in Entwicklungsprozessen. Berichte aus der Informatik. Shaker Verlag, Aachen (2007); Doktorarbeit, RWTH Aachen University
10. Körtgen, A.: Modellierung und Realisierung von Konsistenzsicherungswerkzeugen für simultane Dokumentenentwicklung. Berichte aus der Informatik. Shaker Verlag, Aachen (2009); Doktorarbeit, RWTH Aachen University
11. Schürr, A., Winter, A.J., Zündorf, A.: Visual Programming with Graph Rewriting Systems. In: Proc. of the 11th Intl. IEEE Symposium on Visual Languages (VL 1995). IEEE Computer Society, Los Alamitos (1995)

12. Nagl, M., Marquardt, W.: SFB 476 IMPROVE: Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik. In: Jarke, M., Pohl, K., Pasedach, K. (eds.) Informatik als Innovationsmotor (GI Jahrestagung 1997). Informatik Aktuell, pp. 143–154. Springer, Heidelberg (1997)
13. COMOS PT: Documentation, http://www.comos.com (2010)
14. Aspen-Technology: Aspen Plus Documentation (2009), http://www.aspentech.com
15. Becker, S.M., Wilhelms, J.: Integrationswerkzeuge in verfahrenstechnischen Entwicklungsprozessen. Verfahrenstechnik 36(6), 44–45 (2002)
16. Körtgen, A., Becker, S.M., Herold, S.: A Graph-Based Framework for Rapid Construction of Document Integration Tools. Integrated Design and Process Science, 21 (2007)
17. Körtgen, A.: Tools for Consistency Management between Process Design Products. In: Proc. of the 8th World Conf. on Chemical Engineering, WCCE 2009 (2009)
18. Becker, S.M., Haase, T., Westfechtel, B.: Model-based a-posteriori integration of engineering tools for incremental development processes. Software and Systems Modeling (SoSyM) 4(2), 123–140 (2005)
19. Becker, S.M., Westfechtel, B.: UML-based Definition of Integration Models for Incremental Development Processes in Chemical Engineering. Integrated Design and Process Science: Transactions of the SDPS 8(1), 49–63 (2004)
20. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: Building interactive tools for visual languages. In: Proc. of the 6th World MultiConf. on Systemics, Cybernetics, and Informatics (SCI 2002), USA. Information Systems Development I, vol. I, pp. 17–22 (2002)
21. Becker, S.M., Lohmann, S., Westfechtel, B.: Rule Execution in Graph-Based Incremental Interactive Integration Tools. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 22–38. Springer, Heidelberg (2004)
22. Schürr, A., Winter, A., Zündorf, A.: The PROGRES Approach: Language and Environment, vol. 2, pp. 487–550. World Scientific, Singapore (1999)
23. Becker, S.M., Herold, S., Lohmann, S., Westfechtel, B.: A Graph-Based Algorithm for Consistency Maintenance in Incremental and Interactive Integration Tools. Software and Systems Modeling (SoSyM) 6(3), 287–315 (2007)
24. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Fundamenta Informaticae 26(3/4), 287–313 (1996)
25. Becker, S.M., Körtgen, A., Nagl, M.: Tools for Consistency Management between Design Products. In: Nagl, M., Marquardt, W. (eds.) Collaborative and Distributed Chemical Engineering. LNCS, vol. 4970, pp. 696–710. Springer, Heidelberg (2008)
26. Körtgen, A., Heukamp, S.: Correspondence Analysis for Supporting Document Re-Use in Development Processes. In: Proc. of the 12th World Conf. on Integrated Design & Process Technology (IDPT 2008), SDPS, pp. 194–205 (2008)
27. Körtgen, A.: Modeling Successively Connected Repetitive Subgraphs. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 428–443. Springer, Heidelberg (2008)
28. Miller, G.A.: WordNet: a Lexical Database for English. Communications of the ACM 38(11), 39–41 (1995)
29. Comos Industry Solutions GmbH: Homepage, http://www.comos.com (2010)
30. Nagl, M. (ed.): IPSEN 1996. LNCS, vol. 1170. Springer, Heidelberg (1996)
31. Becker, S.M., Westfechtel, B.: Integrationswerkzeuge für verfahrenstechnische Entwicklungsprozesse. In: Engineering in der Prozessindustrie. VDI Fortschritt-Berichte, vol. 1684, pp. 103–112. VDI-Verlag (2002)
32. Spanoudakis, G., Zisman, A.: Inconsistency Management in Software Engineering: Survey and Open Research Issues. In: Chang, S.K. (ed.) Handbook of Software Engineering and Knowledge Engineering, vol. 1, pp. 329–380. World Scientific Publishing Co., Singapore (2001)

33. Enders, B.E., Heverhagen, T., Goedicke, M., Tröpfner, P., Tracht, R.: Towards an Integration of Different Specification Methods by Using the ViewPoint Framework. Transactions of the SDPS 6(2), 1–23 (2002)
34. Finkelstein, A., Kramer, J., Goedicke, M.: Viewpoint oriented software development. In: Intl. Workshop on Software Engineering and Its Applications, pp. 374–384 (1990)
35. Taentzer, G., Koch, M., Fischer, I., Volle, V.: Distributed Graph Transformation with Application to Visual Design of Distributed Systems. In: Handbook on Graph Grammars and Computing by Graph Transformation: Concurrency, Parallelism, and Distribution, vol. 3, pp. 269–340. World Scientific Press, Singapore (1999)
36. Wagner, R., Giese, H., Nickel, U.A.: A plug-in for flexible and incremental consistency mangement. In: Proc. of the Intl. Conf. on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-Based Software Development) (2003)
37. Gerber, A., Lawley, M., Raymond, R., Steel, J., Wood, A.: Transformation: The missing link of MDA. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 90–105. Springer, Heidelberg (2002)
38. Kent, S., Smith, R.: The Bidirectional Mapping Problem. ENTCS 82(7) (2003)
39. Baresi, L., Mauri, M., Pezzè, M.: PLCTools: Graph transformation meets PLC design. ENTCS 72(2) (2002)
40. de Lara, J., Vangheluwe, H.: Computer Aided Multi-Paradigm Modelling to Process Petri-Nets and Statecharts. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 239–253. Springer, Heidelberg (2002)
41. Königs, A.: Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation. PhD thesis, Technische Universität Darmstadt, Fachbereich Elektrotechnik und Informationstechnik (January 2009) (dissertation)
42. Wagner, R.: Inkrementelle Modellsynchronisation. PhD thesis, Universität Paderborn, Institut für Informatik, Fachgebiet Softwaretechnik (2009) (dissertation)
43. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
44. Königs, A., Schürr, A.: Multi-Domain Integration with MOF and extended Triple Graph Grammars [online]. In: Language Engineering for Model-Driven Software Development. Dagstuhl Seminar Proc., Dagstuhl, Germany, vol. 04101. IBFI (2005)
45. Van Der Straeten, R., Mens, T.: Incremental Resolution of Model Inconsistencies. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 111–126. Springer, Heidelberg (2007)
46. Hausmann, J.H., Heckel, R., Sauer, S.: Extended Model Relations with Graphical Consistency Conditions. LNCS, vol. 2460, pp. 61–74. Springer, Heidelberg (2002)
47. Van Der Straeten, R.: Inconsistency Management in Model-driven Engineering: an Approach using Description Logics. PhD thesis, Vrije Universiteit Brussel (2005)
48. Van Der Straeten, R., D'Hondt, M.: Model refactorings through rule-based inconsistency resolution. In: Proc. of the 2006 ACM symposium on Applied computing (SAC 2006), pp. 1210–1217. ACM, New York (2006)
49. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency Management with Repair Actions. In: Proc. of the 25th Intl. Conf. on Software Engineering (ICSE 2003), pp. 455–464. IEEE Computer Society, Los Alamitos (2003)
50. Egyed, A., Letier, E., Finkelstein, A.: Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In: ASE, pp. 99–108. IEEE, Los Alamitos (2008)
51. Egyed, A.: Fixing Inconsistencies in UML Design Models. In: Proc. of the 29th Intl. Conf. on Software Engineering (ICSE 2007), pp. 292–301. IEEE Computer Society, Los Alamitos (2007)

Towards Semantic Navigation in Mobile Robotics

Adam Borkowski, Barbara Siemiatkowska, and Jacek Szklarski

Institute of Fundamental Technological Research, Polish Academy of Sciences,
Pawinskiego 5b, PL-02-106 Warsaw, Poland
abork@ippt.gov.pl
http://www.ippt.gov.pl/~ztiwww/en/DIShome.html

Abstract. Nowadays mobile robots find application in many areas of production, public transport, security and defense, exploration of space, etc. In order to make further progress in this domain of engineering, a significant barrier has to be broken: robots must be able to understand the meaning of surrounding world. Until now, mobile robots have only perceived geometrical features of the environment. Rapid progress in sensory devices (video cameras, laser range finders, microwave radars) and sufficient computational power available on-board makes it possible to develop robot controllers that possess certain knowledge about the area of application and which are able to reason at a semantic level.

The first part of the paper deals with mobile robots dedicated to operate inside buildings. A concept of the semantic navigation based upon hypergraphs is introduced. Then it is shown how semantic information, useful for mobile robots, can be extracted from the digital documentation of a building.

In the second part of the paper we report the latest results on extracting semantic features from the raw data supplied by laser scanners. The aim of this research is to develop a system that will enable a mobile robot to operate in a building with ability to recognise and identify objects of certain classes. Data processing techniques involved in this system include a 3D-model of the environment updated on-line, rule-based and feature-based classifiers of objects, a path planner utilizing cellular networks and other advanced tools. Experiments carried out under real-life conditions validate the proposed solutions.

1 Introduction

At the onset of Mobile Robotics (MR) most researchers believed that the ultimate goal is to develop robots able to act without human guidance. Most early papers and conferences on MR quote Autonomous Mobile Robots as their subject. Despite the cognitive challenge, this attitude turned out to be unjustified by needs of practice. The most spectacular application of mobile robots — the exploration of Mars — was carried out by means of remotely operated rovers [1]. Apart from units that perform very simple tasks, like cleaning a floor [2] or mowing a lawn [3], mobile robots used to-day in practice depend more or less on

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 719–748, 2010.

human assistance. Moreover, a capability to interact with people and an ability to understand their intentions seems to be a prerequisite for further progress in MR [4].

In order to perform any task, a mobile robot must be able to recognise its environment and to move safely inside it. Such an ability is secured by a *navigation system* of the robot. This system solves the following tasks:

1. *Mapping* means building and updating a *map*, i.e. an internal representation of the surrounding world.
2. *Self-localisation* amounts to determining a current *position* of the robot with respect to a certain reference frame.
3. *Path-planning* means generating a collision-free *trajectory* that leads from the current position to the goal position.

Taking into account many spectacular solutions, shown in the literature (compare, e.g., the overview in [5]), it may be considered that the mapping problem has been satisfactorily solved. Contemporary mobile robots equipped with cameras and laser range finders are able to build 2D-maps of quite cluttered and complex in-door environments. Under such circumstances it seems to be possible to add further dimensions to the mapping problem (2.5D, 3D, 4D maps) without changing the following basic premises:

1. The map is built from scratch, without any initial knowledge about the explored environment.
2. The map describes only geometry of the surrounding world.

The first assumption makes the problem cognitively challenging, though less practically oriented. Service robots are supposed to work in particular buildings, or in particular types of buildings. Digital documentation of any building erected contemporary is available in one of the CAD-formats. A scan of an older building can be easily generated by a laser-based geodetic tool, like Riegl System [6].

A rescue robot might need to build a map of a demolished building, but even then the knowledge of the state, preceding the disaster, is likely to be available. Hence, it seems to be reasonable to relax the first premise by assuming that the layout, taken from the documentation of the building, can serve as the initial assumption about the environment. Moving around the building, the robot is able to detect objects that were not shown on the initial map (e.g. pieces of furniture). It might happen sometimes that certain features of the real layout differ from those described in the documentation of the building (e.g. a door may become blocked or even removed). As an outcome of the process of exploration, the robot obtains a current map of its working environment. Taking floor plans provided by the designers of the building as the initial step of the mapping procedure considerably reduces the computational burden. Instead of generating the map from scratch, the robot registers only missing elements and changes in the environment. Such procedure is illustrated by Fig. 1, where the robot detects an object in the room and recognizes it as a wardrobe.

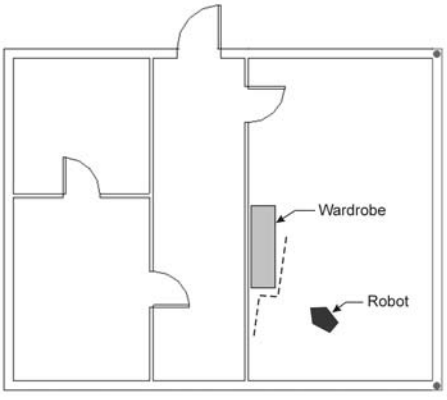


Fig. 1. Exploring a floor of a building

In order to act purposefully in the building, the mobile robot must be aware of the meaning of a surrounding world. A capability of breaking the barrier that separates the acquisition of geometrical features from the understanding of semantics will determine the success of robots in the future. Most researchers recognize this challenge and the first attempts to add meaning to environment maps are reported in the literature [7,8]. These works follow the pattern recognition scheme. It is assumed that the robot is given a certain knowledge about the building (ontology). Such knowledge allows the robot to recognize the components of the layout (windows, doors, etc.) on the current map, or even to detect the functional role of a particular subspace of the building (a hall, a staircase, etc.). Learning semantics autonomously is a difficult task. An obvious solution is to do it interactively with the help of a human operator.

The layout of our paper is as follows. In Section 2 we recall primaries of the geometry-oriented navigation known in the literature, we discuss limitations of this approach and we introduce our proposal of the navigation based upon semantics of the surrounding world. Section 3 is devoted to the new standard of the Architecture-Engineering- Construction (AEC) industry called Building Information Model (BIM). We show that this knowledge representation scheme contains a lot of information useful for in-door class mobile robots. By extracting this information from BIM-files, a preliminary mapping of the building is obtained. This mapping can be validated and enriched by the robot exploring its working area.

In Section 4 we demonstrate how a mobile robot equipped with a laser range scanner can recognise characteristic objects in the environment. Such objects, like a door or a piece of furniture, are stored in the lowest level of a sematic map proposed in Section 2. The path planning by means of Cellular Neural Networks (CNN's) is the subject of Section 5. Here the novelty lies in the mixed metric-symbolic format of the map that is used. A brief summary of results and a list of referencies conclude the paper.

2 Semantic Navigation

Until now most effort in mobile robotics has been spent on *geometric navigation*. Given a start position of the robot S and a goal position G, the navigation system has to find a trajectory T that brings the robot from S to G avoiding obstacles on the way. It is assumed that the positions S and G, as well as the location and shape of obstacles, are given with respect to a certain global reference frame. Under such premises, finding T would be a purely geometrical problem leading to an infinite number of solutions. Therefore, usually additional constraints are imposed on the trajectory, like the minimum length or the minimum energy consumption. This leads to alternative solutions shown in Fig. 2 by a solid line and a dashed line.

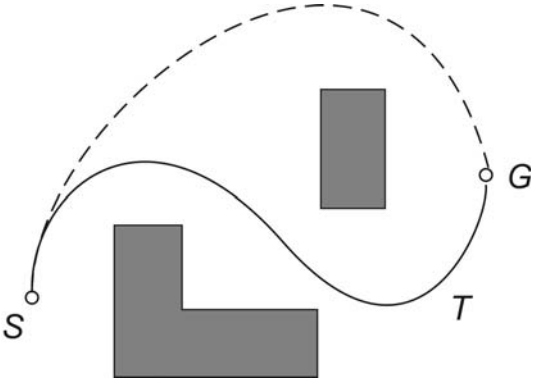


Fig. 2. Main components of geometrical navigation: start position A, goal position B, trajectory T, free space and obstacles

The advantage of the geometric navigation is that it allows the control system to move the robot more or less safely around its working area. The drawback of this procedure lies in the lack of links to the task planning performed at a higher level of abstraction than pure geometry. Let us consider a robot that should perform transportation tasks in an office building. Its current task could be to bring a parcel of books to the library. Assume that this parcel was placed on the robot while it was standing in the hall of the building. Thus, *Hall* is the starting place and *Library* is the goal place. Note that there is no need of geometrical data about start and goal, provided the layout (topology) of the building is given. Let it be a graph depicted in Fig. 3. Nodes of this graph correspond to particular elements of the building. They bear names reflecting functionality of these elements, which simplifies the human-machine interface. The order *Go to the library* can be given via a touch screen or via a natural language recognition system.

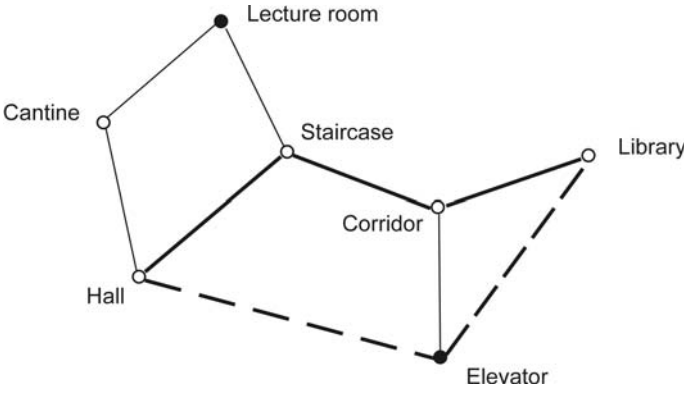


Fig. 3. Main components of semantic navigation: start place *Hall*, goal place *Library*, passable place *Staircase*, impassable place *Elevator*, trajectory {*Hall, Staircase, Corridor, Library*}, alternative trajectory {*Hall, Elevator, Library*}

Edges of the layout graph (usually called topological map) reflect accessibility relations between places. Knowing them, it is relatively easy to generate paths that lead from the starting place to the goal place. Many efficient graph search algorithms can be used for this purpose. Similarly as in the case of geometrical navigation, additional requirements should be posed in order to obtain an unique trajectory. A cost of traversing an edge could reflect either a distance between places or any other parameter. It is also possible to mark nodes of the graph as temporarily passable or impassable. In Fig. 3 the latter are drawn as circles filled with black color. So, despite the circumstance that the trajectory {*Hall, Elevator, Library*} is less "expensive", the robot will go through the staircase and corridor, because the elevator is blocked for the time being.

Representing the layout of the building by a graph seems to be natural but flat graphs are insufficiently expressive. Therefore, we prefer to use multilayered or hierarchical graphs. The usage of such graphs for representing knowledge about layout and functionality of buildings was subjected to our joint study with the group led by Prof. Manfred Nagl [17,18,19,20].

Let us take an exemplary office building as a case study (Fig. 4). Omitting details, we can present its layout in a three-level graph, as depicted in Fig. 5. The top level describes blocks of the building. It is seen that long distance paths require transversing the *Central unit*, where elevators are located. Each floor has similar layout with rooms accessible directly from a corridor. As it will be shown in the following Section, the functionality of a particular place (room) can be stored in an attribute of the object. Such an attribute is rather static in its nature, whereas other attributes (e.g. accessibility or passability) can be time dependent.

The lowest level encompasses a single place. Here nodes of the graph correspond to characteristic objects (landmarks) that can be detected by the means of the sensoric devices mounted on the robot. At the room level we change the

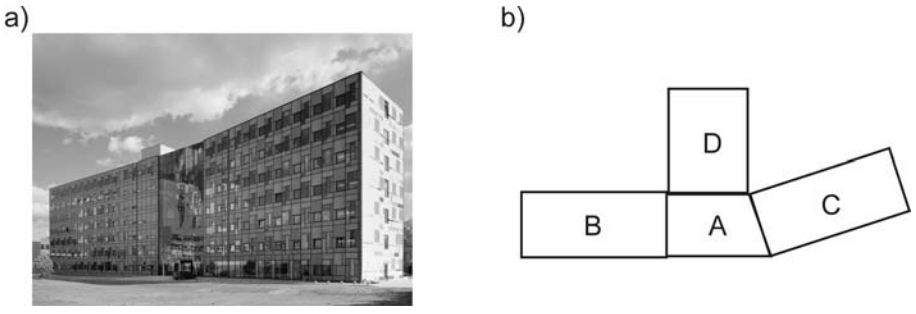


Fig. 4. Exemplary office building: a) overall view; b) footprint

meaning of edges. Instead of accessibility relations attached to edges at higher levels of the graph, we now introduce the global frame of reference *Nord-South, East-West* and store in the edges relative positions of landmarks with respect to this frame. Thus, a door may be situated to the north from the center of the room, the desk to the east, etc. This requires, of course, the robot to be equipped with a digital compass, which is a low cost sensor nowadays.

Within the frame of semantic navigation positioning means knowing the name of a place in which the robot currently is. This can be achieved either by reading a properly placed label (barcode, RFID, etc.) or by recognising the place on the basis of its characteristic features. In the following Sections we show some ways how such recognition can be accomplished.

3 Extracting Semantic Features from Documentation of Buildings

3.1 Data Models in AEC-Industry

The support of 3D modeling of buildings can be traced back to the early 1970s. At first, modeling was based on geometrical primitives, like octahedron, cylinder or pyramid, that were combined by means of Boolean operations (union, inter-section, difference, etc.) to represent more complex shapes (Fig. 6a). Later, Constructive Solid Geometry (CSG) was introduced, defining a final shape via a sequence of Boolean operations ordered in a tree. An alternative way was the so-called Boundary Representation (B-rep). In this model the shape is described by vertices and edges, as shown in Fig. 6b. Both representations have played a useful role in the development of CAD-systems – they have simplified the integration of efforts in different domains of engineering. However, this methodology was not able to represent more than pure geometry of the considered object.

At present AEC industry tries to catch up to aerospace and automotive industries that extensively use digitized models in design, manufacturing and maintenance of their products. It seems that Building Information Modeling (BIM)

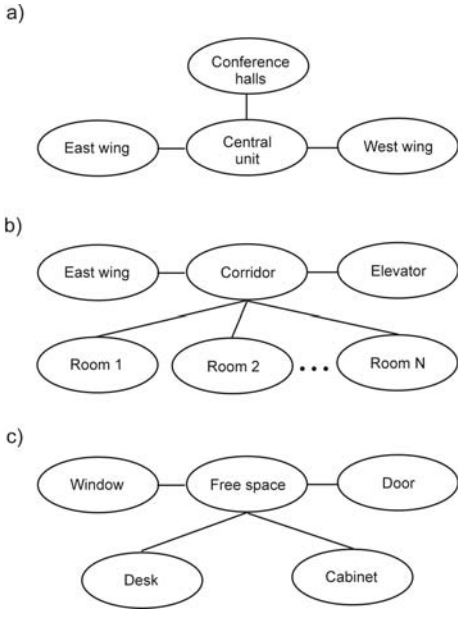


Fig. 5. Topological map of building: a) building level; b) floor level; c) room level

gives a chance of success in this endeavor. The object-based parametric modeling that is the core of BIM-methodology takes the following attitude towards representation of knowledge [9]:

1. Building components are described as hierarchically nested objects that have attributes and application rules.
2. The data structure is consistent, non-redundant, and allows the extraction of multiple views of the object.

A building within the BIM-framework is an assembly of instances of object classes, like walls, floors, ceilings, etc. There is information about how these components have been assembled together and about the constraints that make the design feasible. It is obvious that intelligent components may ,,be told" to take into account the needs of mobile robots as well. For example, a floor could refuse to accept steps and a corridor could check whether it is wide enough for two robots to pass each other.

One of the main advantages of BIM-compliant systems is their ability to interchange data with applications coming from different disciplines. Such interoperability can be achieved using various data formats, but the format Industry Foundation Classes (IFC) [10] seems to be most suitable with respect to the linkage between AEC and MR.

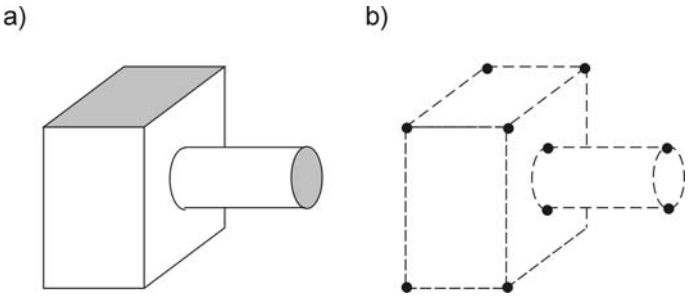


Fig. 6. Defining geometry: a) in CSG style, b) in B-rep style

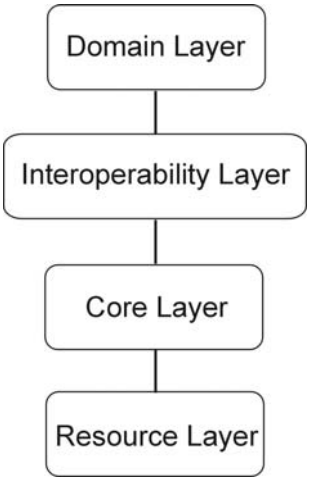


Fig. 7. Main layers of the IFC model

As shown in Fig. 7, the IFC model consists of four hierarchically nested layers. The bottom layer contains 26 reusable constructs like *Geometry, Topology, Materials,* etc. At the level of *Interoperability Layer* these constructs are combined into *Shared Objects*. Such objects include not only generic elements of the building itself, like walls, floors, columns, etc., but also generic elements of the service, management and facility domains. This allows the IFC model to cover the entire life-cycle of the building from its design through construction and usage up to final dismantling.

The top layer refers to specific subdomains of knowledge related to AEC. The most frequently used are the domains *Architecture, Structural Analysis, Heating and Ventilation*. The domains *Building Control* and *Telecommunication* become increasingly important in newly erected buildings.

The IFC model follows the known rules of the object-oriented data representation. It uses the EXPRESS data modeling language [12], developed within the

frame of the ISO-STEP (Standard for the Exchange of Product Model Data) initiative. All objects in EXPRESS are treated as entities subject to enumerations and types. Objects stay in hierarchical parent-child trees with child nodes inheriting properties from their parents. At each level of such a tree user-defined attributes and relations can be introduced, which makes the IFC model extensible and adjustable to various needs.

Let us take the EXPRESS definition of *Space* as an example:

```
ENTITY IfcSpace

SUBTYPE OF (IfcSpatialStructureElement);

InteriorOrExteriorSpace:    IfcInternalOrExternalEnum;
ElevationWithFlooring:      OPTIONAL IfcLengthMeasure;
INVERSE
HasCoverings:   SET OF IfcRelCoversSpaces FOR RelatedSpace;
BoundedBy:      SET OF IfcRelSpaceBoundary FOR RelatingSpace;
END_ENTITY;
```

A space represents an area or volume bounded physically or virtually. Spaces are areas or volumes that provide certain functions within a building. Hence, the description of space entity bears important information for a mobile robot that is supposed to act inside this building.

A space can be either interior or exterior. An interior space is associated with a building storey, whereas an exterior space is associated with a building site. A space may span over several connected subspaces. Therefore, a space group provides for a collection of spaces included in a storey. A space can also be decomposed in parts, where each part defines a partial space. This is defined by the *CompositionType* attribute of the supertype *IfcSpatialStructureElement* which is interpreted as follows:

```
COMPLEX = space group
ELEMENT = space
PARTIAL = partial space
```

The inheritance chain of *IfcSpace* is rather long:

```
IfcRoot - IfcObjectDefinition - IfcObject - IfcProduct -
IfcSpatial-Element - IfcSpatialStructureElement - IfcSpace.
```

Due to this chain IfcSpace obtains several attributes that carry semantic information (*Name, Description, LongName, ObjectType*). The functional category of space is usually stored in the last attribute. Unfortunately, BIM-capable systems available at present are not very good at representing spaces and their assemblies. ArchiCAD [13] allows the user to define *Zones* and Revit Architecture [14] is able to extract automatically *Rooms* as spaces bounded by *Walls, Floors* and *Ceilings*.

An important ingredient of the IFC model is a *Property Set* (P-set). Property sets are defined by the following entity:

```
ENTITY IfcPropertySet

SUBTYPE OF (IfcPropertySetDefinition);

HasProperties:      SET [1:?] OF IfcProperty;
WHERE
WR31: EXISTS(SELF\IfcRoot.Name);
WR32:  IfcUniquePropertyName(HasProperties);
END_ENTITY;
```

The P-set *SpaceCommon* is a property set common for all types of spaces. The exemplary set *SpaceFireSafetyRequirements* contains the fire safety requirements for all types of spaces. When needed, new P-sets can be defined by the user.

A property of an object is defined as:

```
ENTITY IfcProperty

ABSTRACT SUPERTYPE OF   (ONEOF(IfcComplexProperty, IfcSimpleProperty));

Name     :      IfcIdentifier;
Description: OPTIONAL IfcText;
INVERSE
PartOfPset: SET OF IfcPropertySet FOR HasProperties;
PropertyForDependance: SET OF IfcPropertyDependencyRelationship
            FOR DependingProperty;
PropertyDependsOn: SET OF IfcPropertyDependencyRelationship
            FOR DependantProperty;
PartOfComplex: SET OF IfcComplexProperty FOR HasProperties;
END_ENTITY;
```

It is seen from this definition that a property can be either complex or simple. The definition of the latter reads:

```
ENTITY IfcSimpleProperty

ABSTRACT SUPERTYPE OF   (ONEOF(IfcPropertySingleValue,
         IfcPropertyEnumeratedValue, IfcPropertyBoundedValue,
         IfcPropertyTableValue, IfcPropertyReferenceValue,
         IfcPropertyListValue))
SUBTYPE OF (IfcProperty);
END_ENTITY;
```

The above definitions of properties fulfill most needs of AEC industry. A serious drawback with respect to mobile robotic applications is the deterministic nature of properties in the IFC model. Moreover, the present release of IFC

contains a very limited number of properties describing the assumed function of a space, a required security level in a specific zone of the building or other functionality-oriented features. Some proposals how to overcome this deficiency will be given in Section 3.3.

IFC also provides means of expressing relations between objects. These relations are grouped into abstract classes as follows:

1. *Assigns* defines relations between instances and parent entity.
2. *Decomposes* describes assemblies and their parts.
3. *Associates* relates information shared in different parts of the model.
4. *Connects* defines topological relationships between adjacent entities.

It is clear that the present set of relations must be enriched in order to make BIM robot-friendly. This issue is also discussed in Section 3.3.

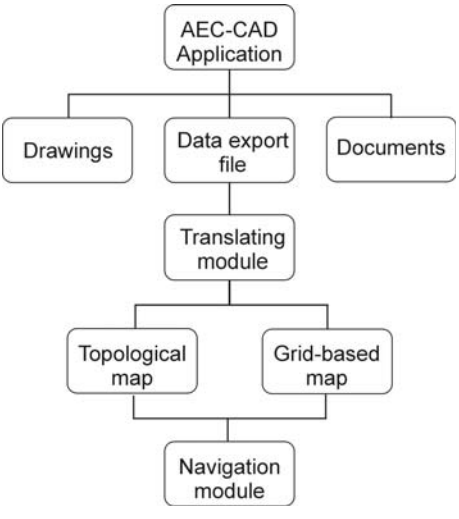


Fig. 8. Data transfer from AEC-format to MR-format

3.2 Taking Advantage of Existing BIM

The attempts to use documentation created when the building was designed for building metric maps applicable in mobile robotics were made several years ago [15,16]. However, at that time BIM was not available and extracting anything more than geometry from CAD drawings was almost impossible. In principle, one could think about representing maps of an in-door environment in one of the data formats popular in the AEC-industry. However, there are too many of them and they are less suited for sensor-based mapping and navigation than formats developed by the MR-community. Therefore, the scheme depicted in Fig. 8 seems to be preferable.

The simplest way of transferring data from one application to another is to use a proprietary file exchange format supplied by a CAD-vendor. We use this procedure in the currently running project, worked up in cooperation with the research group from the Jagiellonian University in Krakow [17].

The aim of this project is to develop a prototype software that translates floor layouts produced by Revit Architecture into three types of maps suitable for mobile robots:

1. Occupancy grids.
2. Accessibility graphs.
3. Feature maps.

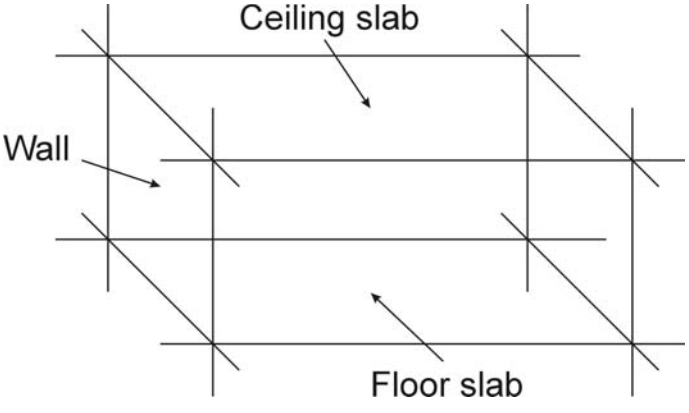


Fig. 9. Room as a subspace of building

Occupancy grids serve as ground knowledge for the mapping module, according to the scheme described in Section 4. Accessibility graphs and feature maps help the navigation module in planning the routes that bring the robot into desired places in the building. At present our translator takes into account only the information present in the standard AEC-oriented BIM model of the building. In such a model a building is composed of *IfcWalls*, and *IfcSlabs* – the entities defined in *IfcShared-BldgElements*. Knowing these entities and relations *Connects* between them, Revit Architecture automatically generates rooms as such spaces bounded by walls, floors and ceilings (Fig. 9).

Rooms are treated as *IfcSpace* entities possibly related by *Connects* relation (adjacency). There is no relation *Accesses* in the presently available BIM model. Fortunately, it is easy to check which pairs of rooms are mutually accessible: they must be adjacent and have a door in a common wall. IFC data sets uniquely define locations of doors and their belonging to particular walls. Hence, a topological map containing adjacency and accessibility relations can be generated relatively easily. Figure 10 shows a simple example of such a map.

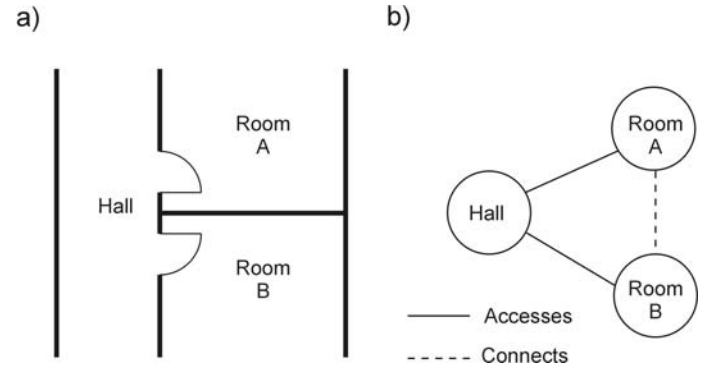


Fig. 10. Adjacency and accessibility relations: a) part of floor layout; b) topological map

The nodes of this graph can be attributed to values taken from P-sets of rooms. A label indicating the functional role of the room is probably the most valuable information for the mobile robot. Such a label could be stored by the designer either in *LongName* or in *ObjectType*. Other attributes, like an area of the room, could also be of interest for a service robot.

Most commercially available AEC-CAD tools support architects, structural engineers and other specialists in the phase of detailed design. Prior to entering this phase, designers must take many conceptual decisions that are crucial for the quality of the building. In our earlier papers [18,19] we investigated how an architect can be supported by a computer in the phase of preliminary design.

It turned out that, at least for a functionality-driven design, a certain formalization of reasoning about the goals, that are to be fulfilled by the designed building, helps the designer to find proper conceptual solutions. Such a formalization can be based upon the theory of graph transformations [20]. It allows the designer to convert informal wishes expressed by the investor into a formal specification of the building. This specification is stored in a form of the functionality graph serving as a framework for various special layouts that are considered in the conceptual phase of design. Functionality graphs bear close resemblance to topological maps of the building. Therefore, we intend to incorporate their usage in the future release of the tools transferring knowledge about the building from the AEC-domain into the MR-domain.

3.3 Adapting BIM for Mobile Robots

Mobile robots can be seen as facilities inside intelligent buildings. The *Domain Layer* of IFC includes *Facilities Management*. Base entities of this domain give the user a possibility to include information relevant for mobile robots into BIM-based description of the building.

Let us consider a typical layout of a control system for an intelligent building (Fig. 11). Such a building has multiple stationary sensors that measure

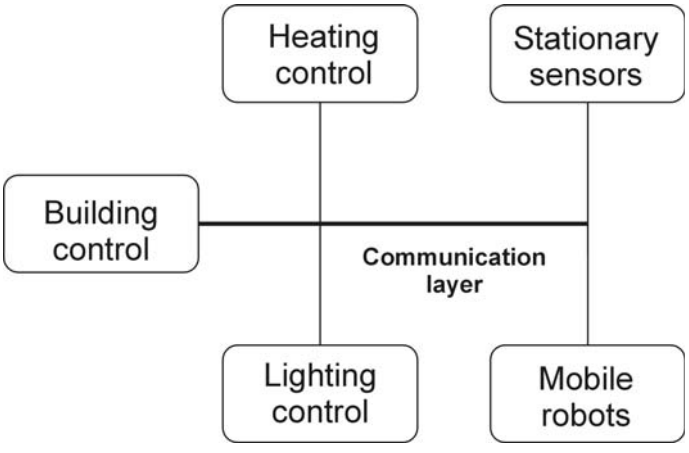


Fig. 11. Mobile robots embedded in an intelligent building

temperature, lighting level and other parameters determining living comfort in particular rooms. The data from the sensors are transmitted via an internal communication network to controllers responsible for heating, ventilation, lighting, etc. The overall state of the building is monitored by the main controller that could take under its supervision mobile robots as well. These robots could perform various tasks, like cleaning floors of the building, transporting goods inside it, etc. In order to be more precise, let us fix our attention on one particular, though quite important task: protecting the building against intruders. The security system could be based upon a combination of stationary sensors and mobile sentry robots. After receiving a signal that a certain suspicious motion has been detected by a sensor in a particular room, the building controller could send a sentry robot to check this room.

Can such a scenario be described by means of BIM? The answer to this question is to a large extent positive. Stationary sensors are quoted in *IfcBuildingControlsDomain*:

```
TYPE IfcSensorTypeEnum = ENUMERATION OF

(   FLOWSENSOR, GASSENSOR, HEATSENSOR, HUMIDITYSENSOR, LIGHTSENSOR,
MOVEMENTSENSOR, PRESSURESENSOR, SMOKESENSOR, SOUNDSENSOR,
TEMPERATURESENSOR, USERDEFINED, NOTDEFINED);
END_TYPE;
```

A possibility to control various facilities of the building is granted by the following entity:

```
ENTITY IfcControl;
. . .
Controls: SET OF IfcRelAssignsToControl FOR RelatingControl;
END_ENTITY;
```

included in the *Facility Management Domain*. Examples of possible actions encompass moving an object inside the building:

```
ENTITY IfcMove

SUBTYPE OF (IfcTask);

MoveFrom:   IfcSpatialStructureElement;
MoveTo:     IfcSpatialStructureElement;
. . .
END_ENTITY;
```

or triggering an action:

```
ENTITY IfcOrderAction

SUBTYPE OF (IfcTask);
. . .
END_ENTITY;
```

when certain conditions hold. Thus, the controller of an intelligent building can be implemented as a rule-based inference engine.

Moreover, BIM introduces *Actors* that play certain *Roles*:

```
ENTITY IfcActorRole;

Role: IfcRoleEnum;
UserDefinedRole: OPTIONAL IfcLabel;
Description: OPTIONAL IfcText;
WHERE

WR1:    (Role <$>$ IfcRoleEnum.USERDEFINED) OR
((Role = IfcRoleEnum.USERDEFINED) AND
EXISTS(SELF.UserDefinedRole));
END_ENTITY;
```

Until now, this part of BIM has served for describing human actors that take part in designing, constructing and exploiting buildings:

```
TYPE IfcRoleEnum = ENUMERATION OF

(   SUPPLIER, MANUFACTURER, CONTRACTOR, SUBCONTRACTOR,
ARCHITECT, STRUCTURALENGINEER, . . .,
USERDEFINED);
END_TYPE;
```

The open character of BIM allows the user to define his or her own actors and their roles. Thus, mobile robots as active elements of intelligent buildings can be easily modeled in the future. Moreover, BIM provides the possibility to define *Views*. These are submodels tailored for the purpose of specific applications. At present only the AEC-oriented view is available. It takes into account needs of the major players in designing conventional buildings: an architect and a structural engineer. A shift towards intelligent buildings will justify the effort to develop a control-oriented view. Such a view should enable architects, developers of mobile robots, suppliers of sensors and software engineers to cooperate efficiently on elaborating complex solutions for comfortable housing.

4 Extracting Semantic Features from Laser Scans

Robotics was defined as *the intelligent connection of perception with action*. A variety of sensing techniques is available to provide the perception. In mobile robotics usually laser range finders are used in order to measure the distance to objects in the operation area of the robot and the odometry to measure internal parameters of the vehicle. The advantage of 2D laser range finders is high accuracy and reliability of the measurements, however they scan the environment in a single plane. Obstacles placed below or above that plane cannot be detected. Today a lot of methods for 3D sensing are based on CMOS/CCD techniques. Typical CMOS/CCD 3D systems are based on the stereo vision and like all passive sensors they have difficulties providing reliable data in an environment with changing illumination. Since 2006 the 3D range cameras have been available. Active methods like 3D laser scanners give a better robustness.

To create a global model of an environment the scans have to be represented in the same coordinate system. This process is called registration. Many representations have been proposed, one of the most popular is 2D representation [21,22,5]. However, when the service robot acts in a domestic environment the 2D-world model assumption is not fulfilled and a 3D method is preferred. Methods of 3D map building can be roughly divided into following groups:

- Full 3D representation of the environment: meshes [23,24], point clouds [25], 3D evidence grids. Methods of this kind allow to represent an unstructured environment very precisely but are computationally expensive and consume a large amount of memory.
- In the second group the environment is represented as a set of features for instance walls [26,27,28]. The advantage of this approach is the compact data representation but it cannot be used directly for a collision-free path planning.
- The third approach combines 3D map and 2D grid-based representation. 2.5D elevation maps, extended elevation maps and multilayer maps are examples of this method [27,29].

The algorithm described in Section 2 belongs to the third group but the map represents not only metric information but also nonmetric properties of the environment. We call this kind of representation semantic map. It means assigning meaning to data. This kind of representation has simplified human-robot communication. For instance the goal for the robot can be given using semantic labels.

The path planning problem is typically formulated as follows: Knowing the goal position and the robot position, find a collision free path leading from the robot to the goal. The path should satisfy certain optimization criteria (for instance it has to be the shortest one). Many path-planning algorithms have been proposed [32], they can be classified according to two factors: the type of the environment (static or dynamic) and the knowledge about the environment (global or local). The global path planning algorithms requires a complete knowledge about the entire environment which is computationally expensive, and a large amount of memory is required. The local path planning algorithms make use of local knowledge only, which is faster, a small amount of memory is needed, it is easier to respond to any local environmental change. However, the solution, e.g., the shortest path, is not necessarily optimal.

Our approach combines the advantages of local and global methods: the planned path is optimal, the path is re-planned in response to changes in the environment, the problem of local minima does not exist. Moreover, it is easy to implement CNN on a parallel architecture in order to increase efficiency.

The scanning system used in our experiments is built on the basis of a SICK LMS 200 laser range finder which is mounted on a rotating support. Sick LMS 200 measures distances to the obstacles in 2D plane with resolution 0.1°, 0.5° or 1.0° and swiping space from −90° to +90°. A plane with 181 (or 361) data points is scanned in 13ms (or 26ms). The laser scanner is mounted on a special support which rotates vertically from −15° to +90°. Depending on the scanning resolution it takes from 100ms to 3s. Figure 12 depicts the scanning system.

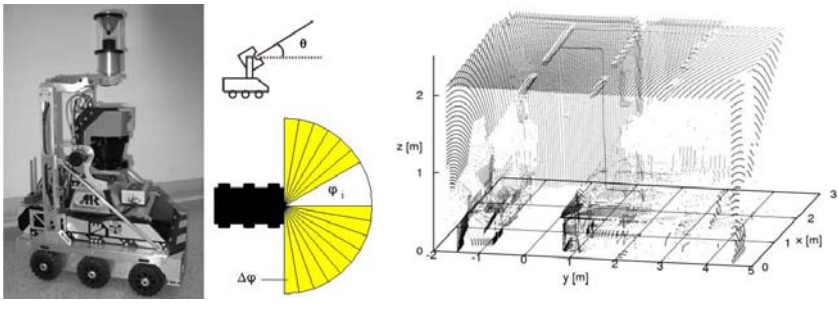


Fig. 12. 3D scanning system: a) Robot Elektron1, b) rotating support and LMS 200 scanning system, c) a sample cloud of points

Data from the laser are obtained in the local polar coordinate system $(r_{i,j}, \phi_i, \theta_j)$, where: $r_{i,j}$ - the distance to an obstacle [meters], θ_i - the vertical angle, ϕ_j - the horizontal angle. The local Cartesian coordinates are computed as follows:

$$\begin{aligned} x_{i,j} &= r_{i,j} \cdot \cos\theta_i \cdot \cos\phi_j, \\ y_{i,j} &= r_{i,j} \cdot \sin\phi_j, \\ z_{i,j} &= r_{i,j} \cdot \sin\theta_i \cdot \cos\phi_j, \end{aligned} \tag{1}$$

where $(x_{i,j}, y_{i,j}, z_{i,j})$ are Cartesian coordinates of the point (i, j).

Usually the next step is to transform these values into a point cloud which is a set of 3D points in the Cartesian coordinate system with the robot in its center (Eq. 1). Such a point cloud can be analyzed for a single 3D scan, or all the points can be combined from different scans to form a global representation of the environment in which the mobile robot is embedded.

For most applications, direct data from point clouds are not sufficient and for some, usually quite sophisticated ones, additional processing is required. Generally, the aim of such processing is to detect and then gather information about specific objects present in the environment. In the field of modern robotics it is usually expected not only to trace the objects of interest, but also to assign some semantic meaning to them.

Often the first step is to find some regular structures in the point cloud: flat or curved smooth surfaces like spheres or cylinders, line segments joining surfaces, etc. Therefore, each point from the cloud can be assigned to its specific element. Algorithms used in this process often have their counterparts in the image analysis field, like the split-and-merge method, surface growing methods, or the generalization of Hough transform [34]. Afterwards a single detected element or a group of connected elements can be recognized as an object. Of course applications of the discussed process are not only limited to robotics. Due to the popularity of high-resolution 3D scanners they are used in architectural modeling [35], industrial applications transforming real scenes into virtual reality [36] and many more. However, it should be noted that it is of vital importance that all the algorithms applied for real-time semantic object detection in mobile robotics are fast.

Below we shortly discuss our simple approach to a very fast scene segmentation with use of standard methods for image analysis. Essentially, the method is based on the direct transformation of a point cloud from a single scene into the RGB image. Afterwards we outline our knowledge-based expert system for semantic labeling detected objects.

4.1 Converting Acquired Data into a 2D Image and Image Segmentation

The most straightforward way to transform data from the laser scanner into an image is to use (ϕ, θ) as the pixel coordinates and assign its color according to the measured distance. A sample scene is shown in Fig. 13.

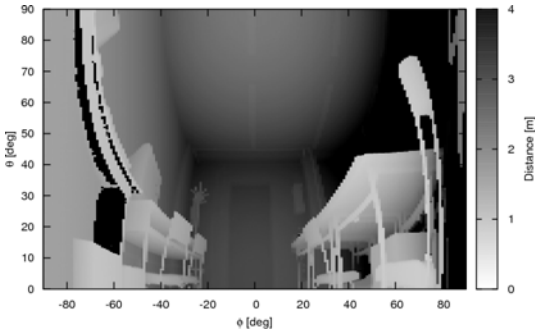


Fig. 13. A sample gray-scale image representing distance from the robot in (ϕ, θ) space. Maximum distance measured by the scanner is 8 meters. (Here, for clarity, we show all distances greater than 4 meters as black.)

As first step of our process a flood fill algorithm is run on the image representing distance. The threshold for the algorithm is constant and it corresponds to about 5 cm (the difference between neighbouring pixels is considered when flooding). All areas which are too small to be classified are marked with number 0 and are not considered in any later stage of the process.

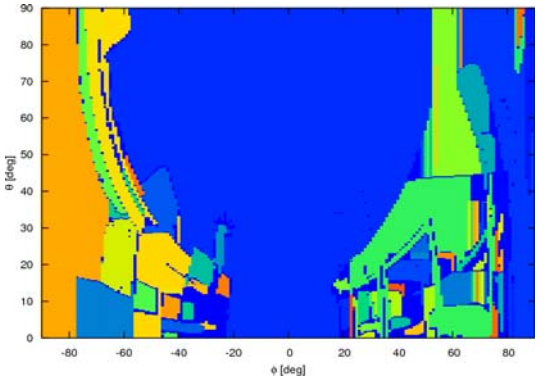


Fig. 14. Areas detected after the first stage of the segmentation process applied to data presented in Fig. 13. Each area is marked with a different, arbitrary chosen, color.

After the first step is finished, one has a list of areas which represent ,,continuous structures" of the environment. For example, a chair standing in front of a wall will be assigned to a different area than a wall since there is a large change of distance between the chair's edge and the wall. On the other hand, walls, ceiling, and floor will be classified as a single area since the change of the distance in all corners is assumed to be small.

In order to obtain images more suitable for the information extraction we map three coordinates associated with a normal vector for each pixel to RGB values of an ordinary color image. For each pixel (i, j) we obtain its position $\mathbf{p}$ in 3D Cartesian coordinates with the robot in its center. Then a normal vector at (i, j) is calculated as a sum of cross products of $\mathbf{p}$ and vectors associated with the four closest neighbours of (i, j).

A color RGB image is constructed by assigning values of the coordinates $\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z$ as colors red, green and blue, accordingly. Then $\mathbf{p}$ is normalized and each coordinate is mapped to an 8-bit color component as $(-1, 1) \rightarrow (0, 255)$. Note that, for example, a ceiling or a floor will have red and green components equal to 128, while the blue one will be larger than 128 for the floor (making it blueish) and smaller than 128 for the ceiling (making it look more yellowish). On the other hand, all of the planes perpendicular to the laser scanner will have the blue component equal to 128. Moreover, walls which are placed along the line of sight of the robot will be pink on their left-hand side and cyan on the other side. An image generated in this way is depicted in Fig. 15.

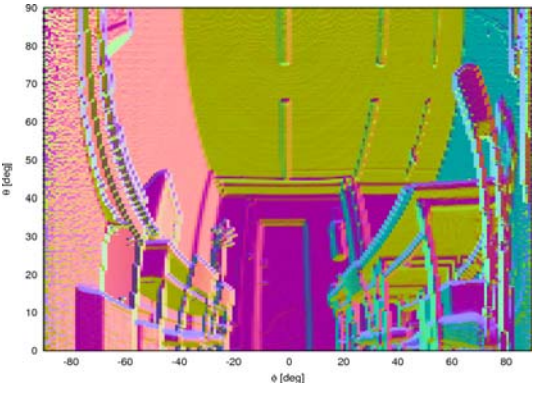


Fig. 15. Data for the scene presented in Fig. 13 were transformed in order to obtain normalized normal vectors $\mathbf{n}$ for each pixel. Absolute values of (x, y, z) coordinates of $\mathbf{n}$ are presented as red, green and blue components of this 8-bit RGB image (e.g., $\mathbf{n}_x = 0$ gives red=0, $\mathbf{n}_z = \pm 1$ gives blue=255). Note the increasing noise for ϕ near $\phi = \pm 90°$.

Naturally, the straightforward mapping of angles ϕ, θ as pixel position does not have to lead to the construction of images optimal for subsequent processing. For example, it is evident that all measurements with $\phi = \pi/2$ (for $\phi = -\pi/2$ as well) represent in fact the same point regardless of the value of θ. Consequently, the resulting image in the region where $\phi \approx \pm\pi/2$ is strongly distorted, also a significant noise is observed. The noise arises due to the method for calculation of the normal vectors – the entire region represents a small area of the real environment and statistical errors from the laser are exaggerated accordingly. Therefore, in our research we consider other mappings as well.

For example it is convenient to convert data in (ϕ, θ) coordinates to the usual spherical coordinate system $(\hat{\phi}, \hat{\theta})$ in which the robot is in the center, $\hat{\phi}$ is the longitude and $\hat{\theta}$ is the latitude. Having done this, it is possible to use, e.g., the Albers equal-area conic projection [37] which produces images representing the environment in a somehow more natural way. Obviously, one has only a limited number of measured points and therefore it is necessary to apply the appropriate interpolation when projecting from (ϕ, θ) coordinates.

The purpose of the discussed segmentation is to perform a *fast* decomposition of the gathered data into areas, each one representing a flat polygon in the real scene. Along the most important areas are, of course, a ceiling, a floor, walls, doors, etc. Besides the list of polygons, some numbers characterizing physical properties of a polygon can also be extracted. These can be used later for better object classification.

Having obtained a list of n areas by running the flood-fill algorithm on an image representing distance, Figs. 13, 14, the next step is performed. It consists of dividing the n areas into smaller ones by running the flood-fill algorithm on the RGB image constructed from normals. For each area i, $1 \leq i \leq n$, it is used as a mask, so only pixels belonging to the area i are used as seed points and all pixels from area outside i act as borders for the filling process. Eventually, the second step gives a list of $m \geq n$ areas each representing a more or less flat surface of the real scene.

In the final phase of the segmentation process the areas from step two are converted into polygons in the full three-dimensional space. For each area i, $1 \leq i \leq n$, we consider its pixels belonging to the *inner* edge of the area on the border between i and the rest of the image. Of course every pixel directly corresponds to a point in 3D space from the point cloud. Taking every 3D point from all pixels forming the edge we obtain a list of vertices defining a connected component – polygon – related to the area i. These polygons are used in a classification procedure.

After the segmentation process is finished, some geometric features of the scene are extracted and a list of flat surfaces is obtained. In order to assign a semantic feature to such a surface we use the following characteristics (see also, e.g., [34]).

- Size – usually an object which is supposed to be detected is characterized by some reasonable geometrical size.
- Orientation – for example: walls or doors are always vertical, the ceiling is always horizontal.
- Topology – relations between surfaces is important.

Based on the above characteristics a simple rule-based classifier is applied. In such a system an object is labeled as *door* if it is a single, vertical surface with the width within the range 1-2m, the height 1.5-2.5m, it connects directly to a *floor* at its bottom and to a *wall* from all other sides.

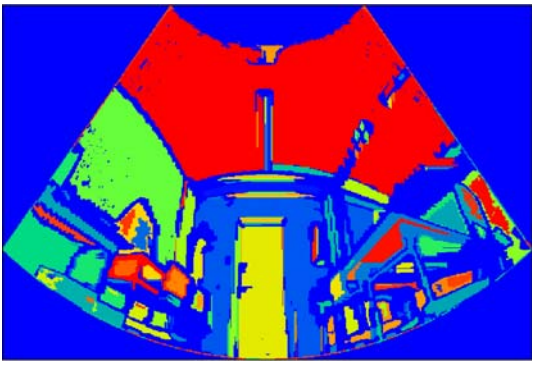


Fig. 16. Areas detected after the segmentation process. However, here processing is done on an image which has been firstly projected using Albers conic projection.

4.2 Rapid Object Detection Using Haar-Like Features

Treating laser scanner data as an image makes it also possible to directly apply well known methods for object detection and pattern recognition from an image analysis field. Here we show how to enrich our classification system by using a scheme based on a boosted cascade of simple features to detect objects. Algorithms which are applied are available in the OpenCV library and they implement methods proposed by [38] for basic set of Haar-like features and by [39] for rotated features which enhance the basic set. After the system is trained for recognition of certain objects, new images can be analysed very fast while maintaining a good hit rate and a reasonably low false alarm rate. This makes the method interesting and practical for our purposes. Moreover, having direct geometrical information about a detected object we can often reject false classifications just by analysing its real size.

Images generated from laser data in the way proposed above have, of course, different properties than usual visual images gathered by cameras. For example, illumination and any lighting conditions are not of our interest here. Either in bright light with many shadows or in a completely dark room one gets the same image. On the other hand, when the robot equipped with the scanner moves on a floor, red and green components of the image change so there is some dependency upon its position. This, however, does not have to lead to problems with the object detection with use of the image analysis, since many of the methods used for that purpose operate on gray-scale images.

In the first stage of our classifier it is necessary to train it for objects of interest. Here we show examples how to detect a sink visible from the perspective of the robot. In order to perform training a Haar-like classifier one needs a large set of positive and negative example images. All of the images should be scaled to the same size, we use 20x20. As the set of negative examples we use a large number of arbitrary images representing a scene without any object of our interest, i.e., without any image of the sink in this case.

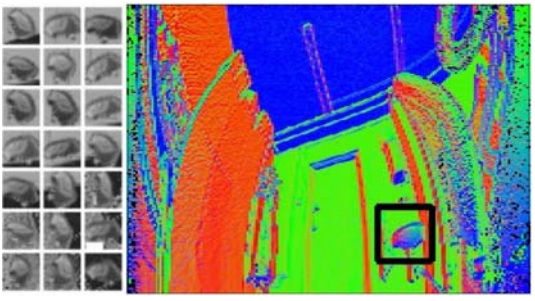


Fig. 17. The left panel shows 21 images – a small subset of set of positive samples which are used to train classifier for a sink-like object. On the right the result of classification for test image is presented (the black rectangle). The test image was not used in the training process. Here when converting the normals to RGB absolute values $|\mathbf{n}_x|, |\mathbf{n}_y|, |\mathbf{n}_z|$ instead of $\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z$ were used.

In order to get a set of positive examples we take several snapshots with the laser scanner of a scene with the sink. Then, after constructing images representing the scene, we crop a sub-image with a sink only, making the background translucent. In the next step all the sub-images are rotated about x, y, z axis by random angles, intensity is randomly modified and sinks are placed onto a random background image. Finally, we have 1000 different images of a sink with random transformations placed onto different backgrounds. Large images containing a sink serve as testing images after training.

After the training process is completed, the classifier can be applied to any region of an image giving *true* if the region is likely to contain a pattern similar to one of those from the positive samples set, *false* otherwise. The analysis is very fast so one can try many different regions with varied sizes from all parts of the image. By doing this in a loop one can search the entire image for an object of interest. Figure 17 shows the result of such an analysis when searching with a classifier trained for the detection of objects similar to a sink and stairs respectively.

Areas corresponding to objects detected with the discussed method can be processed later in our classification scheme when building a semantic map. Naturally, each object which is going to be recognized has to have its own specifically trained Haar-features classifier. In a more sophisticated approach it is possible to combine object recognition from both: visual images and images constructed from laser scans.

Figure 18 depicts a top-view of a sample global 3D map of the environment. In such a map each cell corresponds to a cube of size 10 cm x 10 cm x 10 cm in reality. If a certain object has been detected, appropriate cells in the map are marked with specific semantic labels. On this sample map the following objects have been detected: floor, ceiling (both omitted in the figure), wastebasket, door, stair and washbasins.

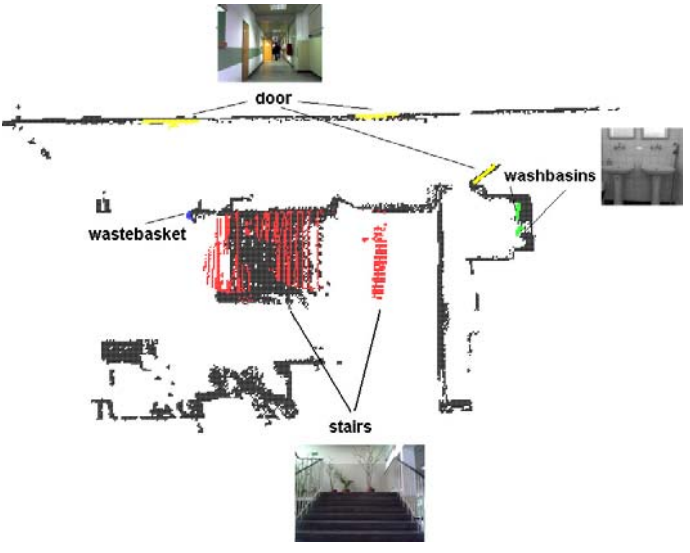


Fig. 18. A top-view of a sample global map of the environment. Cells represent unidentified and identified obstacles, floor and ceiling on the image are omitted.

5 Path Planning

A task planning for mobile robots usually relies on the spatial information. Although this kind of information is necessary for performing basic operations, the use of semantic knowledge is useful at a higher degree of autonomy and intelligence. In this section we show how this type of knowledge can be profitably used for the robot path planning. We start by defining a specific type of semantic maps, which integrate the metric information and the semantic knowledge.

The map of the environment is represented as a grid of cells, a list of semantic labels is attached to each cell. This kind of representation allows us to find easily the position of a specified object in the environment, and the relationship between objects. We can also ask the robot to move to a door or along a corridor. The path-planning algorithm which is proposed in this paper is implemented using the Cellular Neural Network.

5.1 Cellular Neural Network

The Cellular Neural Network (CNN, also known as systolic arrays) was proposed by L. O. Chua in 1988, as a very efficient tool for image analysis [30,31]. CNN consists of neurons (cells) which interact locally. Usually cells are arranged in a form of an $N \times M$ array. The cell in strip i and column j is denoted by the symbol c_{ij}, its state is described by symbol x_{ij} .

The cell c_{kl} belongs to the neighbourhood of c_{ij}, if for a parameter r (radius of neighbourhood) the following condition is fulfilled:

$$\max(|i-k|, |j-l|) \leq r. \tag{2}$$

The neighbourhood of the cell c_{ij} is denoted by the symbol N_r^{ij}. The CNN is defined by the following parameters: input signals $u_{ij}^{kl} \in R$, output signals $y_{ij}^{kl} \in R$, a bias $I \in R$, a_{ij}^{kl} - interconnection weight between the cells c_{kl} and c_{ij}, b_{ij}^{kl} - the feed forward template parameter. The dynamics of the CNN is described as follows:

$$x_{ij}(t+1) = \sum_{c_{kl} \in N_r^{ij}} a_{ij}^{kl} y_{kl}(t) + \sum_{c_{kl} \in N_r^{ij}} b_{ij}^{kl} u_{kl}(t) + I, \tag{3}$$

$$y_{ij}(t+1) = f(x_{ij}(t+1)), \tag{4}$$

were: f is an output function. Chua extended the definition of CNN. It is assumed that CNN consists of cells that interact locally and can be modelled as locally connected finite state machines. The new definition allows to widen the area of possible applications.

5.2 Path Planning

The algorithm for path planning consists of the following parts:

- The $N \times M$ grid-based traversability map is created based on the dual map of the environment, each cell of the map represents the traversability level (u_{ij}) of a corresponding area.
- The $N \times M$ CNN is built, each cell of the map corresponds to the cell of CNN.
- The weights a_{ij}^{kl} of the interconnection between c_{ij} and c_{kl} are computed. a_{ij}^{kl} is proportional to the distance between centres of gravity of areas which are represented by cells c_{ij} and c_{kl}.
- u_{ij} is the input signal to the cell c_{ij}, the value of u_{ij} is computed based on semantic knowledge.
- A set c_G which represents the position of the goal is distinguished. Usually this set consists of more than one neuron, the neuron c_R - represents the position of the robot.
- the value f_{ij} is computed for each cell of CNN:

$$f_{ij}^r = f_o(u_{i-r,j-r}, ..., u_{ij}, u_{i+r,j+r}), \tag{5}$$

  where r is the radius of the neighbourhood, f_o is the function of input signals.
- The state of the cell c_{ij} is described as follows:

$$x_{ij}(t) = \begin{cases} K - f_{ij}^r & if\ c_{ij} = c_G, \\ \max(\max_{kl \in N_2^{ij}}(y_{kl}(t) - a_{ij}^{kl}) - f_{ij}^r, 0) & if\ c_{ij} \neq c_G. \end{cases} \tag{6}$$

If a cell represents the goal position and is free of the obstacles, then its state is static and equals K (very large number). If it is occupied then its state equals to 0 . If a cell is not a goal and is free from the obstacles then its state equals to the maximum value of the neighbouring cells minus the traversing cost in other cases the state equals to 0.

– The output signal y_{kl} of the cell c_{kl} is described as follows:

$$y_{kl}(t) = f(x_{kl}(t)). \tag{7}$$

In our approach $f(x) = x$.

The process of diffusion is continued until stability

$$\forall_{i=1,..,N,j=1,..,M} \quad x_{ij}(t) = x_{ij}(t+1). \tag{8}$$

If the cell c_{ij} represents the position of the robot then the next position of the vehicle is indicated by the neuron $c_{kl} \in N_2^{ij}$ for which the following formula is fulfilled:

$$x_{kl}(t) = \max(\{x_{nm}(t)\}), \text{ where } c_{nm} \in N_2^{kl} \tag{9}$$

The planned path (an ordered list of cells) depends on the values of input signals u_{ij}, function f_o and the radius of the neighbourhood.

In most of the algorithms of path-planning, the robot is represented as a point and the obstacles are extended by some radius in order to take into account the dimensions of the robot. In many practical situations it is difficult to indicate the best value of the radius and the method fails when a group of robots with different sizes plan their paths based on the same map. In our approach, the dimension of the robot is taken into account during the path planning - values of parameters b_{ij}^{kl} are computed. b_{ij}^{kl} indicates the influence of an obstacle in the area represented by the cell c_{ij} to the cell c_{kl}.

If, for instance, the task for the robot is to move to the nearest table, then all cells which correspond to the position of any table are activated and the path is planned automatically. If the task for the robot is to move to the specified table, then the cells which correspond to this table are activated.

Figure 19 presents the result of the path plannig to a washbasin. The dotted line indicates the shortest path and the continuous line referes to the safest one. R - represents the initial position of the robot.

In order to determine low-level primitives, a set of pairs (v_t, ω_t), where v_t is the linear velocity and ω_t the rotational velocity of the robot at the moment t, a modified dynamic window approach is used [33]. In this method the search space consists of velocities that can be achieved by the robot, given its current linear and angular velocities and its acceleration capabilities, within a given time interval. This time interval corresponds to a servo tick of the control loop. In the case of semantic navigation the set of admissible velocities depends also on the environment. The maximal velocity of the robot is large in a big, empty environment but the robot has to slow down near a door.

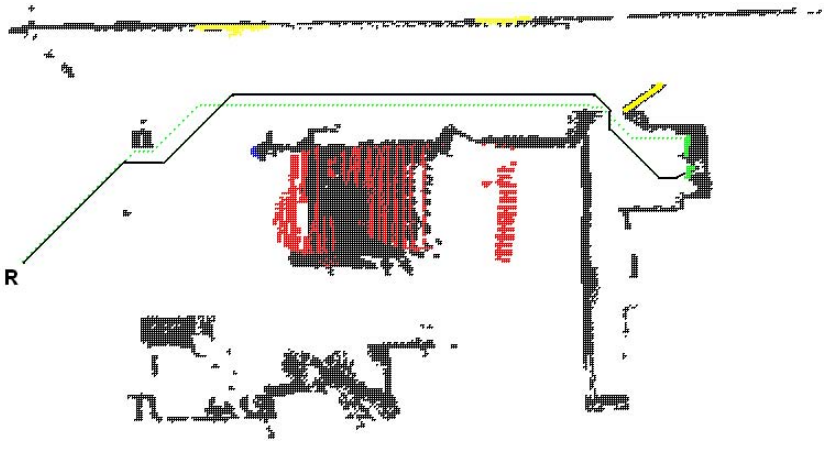


Fig. 19. The path of the robot: the shortest path - dotted line, the safest path - continuous line

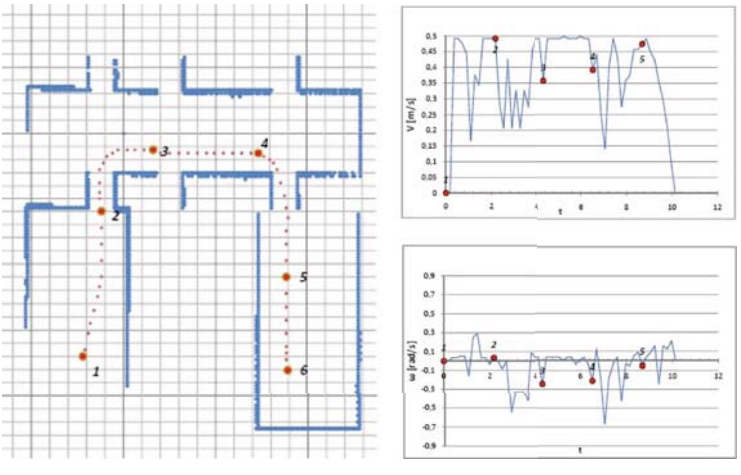


Fig. 20. The executed path of the robot: *Left:* the path of the robot, *Right, top:* linear velocities, *Right, bottom:* angular velocities

Velocities which maximize the function $G(v, \varphi)$ are taken as optimal values.

$$G(v, \omega) = a \cdot \mathrm{Dyf}(v, \omega) + b \cdot \mathrm{Dyf}_{\alpha}(v, \omega), \tag{10}$$

where $\mathrm{Dyf}(v, \omega)$ is the difference between the state value of a cell which represents the current position of the robot and the value of the state after time Δt if controls (v, ω) are applied, $\mathrm{Dyf}_{\alpha}(v, \omega)$ is a function which favours trajectories that lead straight towards the cell (c_{max}). c_{max} has the maximum state value in

the neighbourhood of the cell which corresponds to the current position of the robot, parameters a and b are scaling.

6 Conclusions

The mobile robotics is on the verge of transition from cognitive-driven research to full scale applications. In the foreseeable future in-door class robots will become important ingredients of facilities provided by public domain infrastructure and by newly erected living habitats. In order to achieve the best performance, mobile robots must be fully integrated into the functional concept of the building and the building must be designed taking into account the presence of robots. Such synergy requires the opening of a new interdisciplinary domain of research, where people interested in designing buildings and mobile robots could meet and exchange their needs and ideas. The present paper can be seen as a modest step toward this goal.

Acknowledgement. This work has been partially supported by the Polish Ministry of Science and Higher Education (grant: 4311/B/T02/2007/33).

References

1. Mars Exploration Rover Mission. Jet Propulsion Laboratory, California Institute of Technology (October 3, 2009), http://marsrovers.nasa.gov/home
2. Household Robots. iRobots (October 4, 2009), http://www.irobot.com/uk/home_robots.cfm
3. Robomow by Friendly Robotics (July 22, 2008), http://www.friendlyrobotics.com
4. Fox, D.: Toward High-level Reasoning for Autonomous Systems. In: Lilienthal, A., Petrovic, I. (eds.) Proc. of the 4th European Conference on Mobile Robots (ECMR 2009), Mlini-Dubrovnik, Croatia, pp. 1–6 (2009)
5. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. MIT-Press, Cambridge (2005)
6. RIEGL Laser Measurements Systems (2008), http://www.riegl.co.at
7. Martinez Mozos, O., Triebel, R., Jensfelt, P., Rottmann, A., Burgard, W.: Supervised semantic labeling of places using information extracted from sensor data. Robotics and Autonomous Systems 55(5), 391–402 (2007)
8. Vasudevan, S., Harati, A., Siegwart, R.: A Bayesian Approach to Conceptualization and Place Classification: Using the Number of Occurrences of Objects to Infer Concepts. In: Burgard, W., Gross, H.-M. (eds.) Proc. of the 3rd European Conference on Mobile Robots (ECMR 2007), Freiburg, Germany, pp. 1–6 (2007)
9. Eastman, C., et al.: BIM Handbook. A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors. Wiley, Hoboken (2008)
10. BuildingSMART (former International Alliance for Interoperability IAI), Industry Foundation Classes, Version 2X4 (December 14, 2008), http://www.iai-international.org/index.html
11. Industry Foundation Classes, Release 2x, Platform Specification (IFC2x Platform) (2005), http://www.iso.org/iso/catalogue_detail.htm?csnumber=38056

12. Schenck, D.A., Wilson, P.R.: Information Modeling the EXPRESS Way. Oxford University Press, New York (1994)
13. ArchiCAD, ArchiCAD 12 - Accelerating the Design Experience (December 14, 2008), http://www.graphisoft.com/products/archicad/ac12
14. Autodesk (2008). Revit Architecture (December 14, 2008), http://usa.autodesk.com/adsk/servlet/index?id=3781831&siteID=123112
15. Fennema, C., et al.: Model-Directed Mobile Robot Navigation. IEEE Transactions on Systems, Man and Cybernetics 20, 1352–1369 (1990)
16. Murarka, A., Kuipers, B.: Using CAD Drawings for Robot Navigation. IEEE Transactions on Systems, Man and Cybernetics 2, 678–683 (2001)
17. Borkowski, A., Grabska, E., Palacz, W.: Modeling Buildings for Mobile Robots. In: Proc. ASCE International Workshop on Computing in Civil Engineering, Austin, Texas (2009)
18. Borkowski, A., Grabska, E.: Converting function into object. In: Smith, I.F.C. (ed.) EG-SEA-AI 1996. LNCS, vol. 1454, pp. 434–440. Springer, Heidelberg (1998)
19. Borkowski, A., Szuba, J.: Graph transformations in architectural design. Computer Assisted Mechanics and Engineering Sciences (CAMES) 10, 93–109 (2003)
20. Kraft, B., Nagl, M.: Visual Knowledge Specification for Conceptual Design: Definition and Tool Support. Advanced Engineering Informatics 21, 67–83 (2007)
21. Leonard, J.J., Durrant-Whyte, H.F.: Direct Sonar Sensing for Mobile Robot Navigation. Kluwer Academic Publishers, Boston (1992)
22. Elfes, A.: Sonar-based real-world mapping and navigation. IEEE Transactions on Robotics and Automation, 249–265 (1987)
23. Sakas, G., Hartig, J.: Interactive visualization of large scalar voxel fields. In: Vizualization, pp. 29–36 (1992)
24. Schroeder, W., Zarge, J., Lorensen, W.: Decimation of triangle meshes. Computer Graphics, 65–70 (1992)
25. Rusu, R.B., Marton, Z.C., Blodow, N., Dolha, M., Beetz, M.: Towards 3d point cloud based object maps for household environment. Journal of Robotics and Autonomous Systems 56, 927–941 (2008)
26. Weingarten, J., Siegwart, R.: EKF-based 3D SLAM for structured environment reconstruction. In: Proc. of IROS 2005 (2005)
27. Triebel, R., Pfaff, P., Burgard, W.: Multi-level surface maps for outdoor terrain mapping and loop closing. In: Proc. of IROS, pp. 1–2 (2006)
28. Siemiatkowska, B., Gnatowski, M., Chojecki, R.: Cellular neural networks in 3d laser data segmentation. In: 9th WSEAS International Conference on NEURAL NETWORKS, pp. 84–88 (2008)
29. Gu, J., Cao, Q., Huang, Y.: Rapid traversability assesment in 2.5d grid based map on rough terrain. Internationl Journal of Advanced Robotic Systems 5(4), 389–394 (2008)
30. Chua, L., Young, L.: Cellular Neural Networks. IEEE Transaction on Circuit System 2, 985–988 (1988)
31. Chua, L., Wu, C.W.: On the Universe of Stable Cellular Neural Networks. IEEE Transaction on Circuit System 42, 559–577 (1995)
32. Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion - Theory, Algorithms, and Implementations. MIT-Press, Cambridge (2005)
33. Fox, D., Burgard, W., Thrun, S.: The dynamic window approach to collision avoidance. IEEE Robot. Autom. Mag. 4, 23–33 (1997)

34. Gorte, B., Vosselman, G., Sithole, G., Rabbani, T.: Recognizing structure in laser scanner point clouds. In: Proceedings of Conference on Laser Scanners for Forest and Landscape Assessment and Instruments (2004)
35. Pu, S.: Vosselman, G.: Knowledge based reconstruction of building models from terrestrial laser scanning data. ISPRS Journal of Photogrammetry and Remote Sensing, 0924-2716 (2009) (in Press) (Corrected Proof)
36. Yu, Y., Ferencz, A., Malik, J.: Extracting objects from range and radiance images. IEEE Transactions on Visualization and Computer Graphics 7, 351–364 (2001)
37. Snyder, J.P.: Map Projections – A Working Manual, United States Government Printing, Washington, DC, pp. 98–103 (1987)
38. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: IEEE CVPR (2001)
39. Lienhart, R., Maydt, J.: An extended set of haar-like features for rapid object detection. In: IEEE ICIP 2002, vol. 1, pp. 900–903 (2002)

Model Driven Engineering in Operative Industrial Process Control Environments
- Overview -

Ulrich Epple

Lehrstuhl für Prozessleittechnik, RWTH Aachen
Turmstr. 64, 52064 Aachen
epple@plt.rwth-aachen.de

Abstract. Industrial process control is a profitable field for model driven approaches. The paper presents an overview and discusses the basic concepts and architectural principles. In future an increasing demand for engineering and reengineering activities in the operative phase is expected. In the industrial process control environment the necessary software changes in the operative phase often have to be performed without stopping the execution of the control system. The paper presents ideas and concepts to improve the model driven approaches and outlines ways to an assisted and partially automated control engineering.

1 Introduction

In process industry every plant is unique. Even "exact plant copies" are different. Improved functionality, necessary adaptations to actual and local regulations, the usage of new equipment and the deficiency of the old documentation require a complete new engineering project. The reusability of process control software can be seen as being zero. Process control software is written for only one application and under heavy time and cost pressure. Process control software is written by "normal" automation engineers and poorly tested. On the other hand, strong requirements with respect to software quality and functional safety have to be kept. To master this problem the control engineers try to keep things simple. They follow a traditional model-based engineering procedure. Standardized and vendor-supported components are assembled in accordance to a worldwide common design schema. The components are defined one after the other and assembled by connecting them via designated communication associations. Powerful engineering tools support the control engineering process. They offer a comfortable graphical interface, they support a consistent specification of the inter-component communication, they allow the building of macros, they prevent basic design faults, they assist to distribute the specified functionality to the decentralized hardware allocations, they offer integrated simulation functionality and so on. Due to these powerful tools and the simple and intuitive architecture, the engineering of the control software was not stated as a critical problem. But the situation is changing. The rapidly increasing amount of functionality, the

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 749–765, 2010.

required flexibility even in the operational stage, the need of verification and the loss of competence within the engineering teams cannot be equalized by an additional improvement of the classical tools. New approaches are required.

This paper starts with a short presentation of the structure of the production control task and the respective software architecture principles. Currently upcoming new requirements and challenges will be presented. The application of model-based concepts to manage these requirements will be discussed. The paper ends with a vision of the control architecture of the future.

2 Production Control Architecture

Figure 1 shows a so-called P&I Diagram (P&I =Pipe and Instrumentation) of a chemical reactor. This is a typical example of a plant section in the process industry.

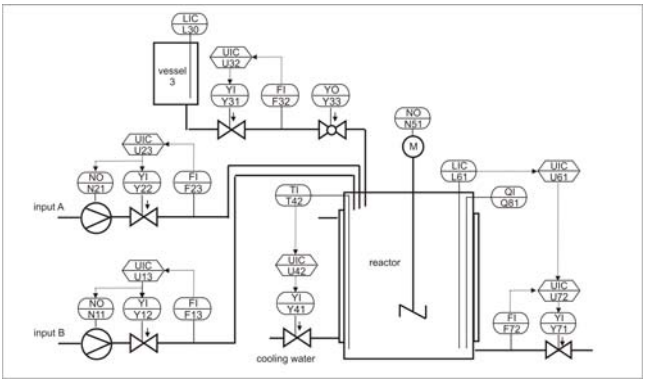


Fig. 1. Example of a P&I-Diagram

In this reactor different chemical processes can be performed. Manufacturing execution functions deal with all aspects of the control and supervision of such type of plants and production processes. In this section we focus on the main production control task. The production control task can be divided into two parts: the plant- or resource-related part and the product- or recipe-related part.

2.1 Plant-Related Control Tasks

In many cases, the basic levels of control are independent of the special production process. In figure 1 for example we find pumps, valves and a stirrer as actuators. As shown in figure 2, for every actuator an individual control module can be specified which captures all control requirements concerning this specific actuator. In a second step we can look for coordination tasks, which are an intrinsic part of the functionality of a plant unit. The feed unit for input *A*

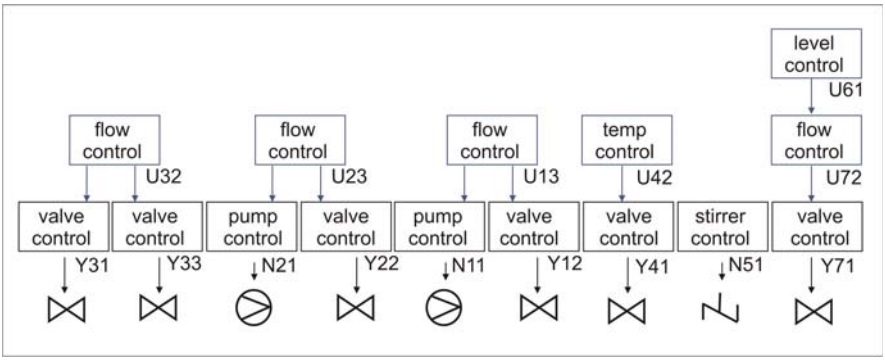


Fig. 2. Plant related control modules

in figure 1 is a typical example. The following plant elements are part of this functional group: on/off-pump $N21$, control valve $Y22$ and the flow meter $F23$. The coordination functions, which are necessary to control the flow-rate or to carry out a dosing of input A, are independent of the specific production process. Following a hierarchical control concept, we can comprise these group control functions in a separate group control module as shown in figure 2. The hierarchical concept assumes that the lower level modules provide complete control services that can be used by the upper level control modules.

On the base of this schema we can implement a bottom-up approach for the specification of the control task. Starting with the single unit level we can specify the actuator control modules. In the next step we define the group control modules, then the section control modules and so on. We can go further as long as we find control functions, which are intrinsic parts of the technological plant and we have to stop if the specification of a functional interrelationship restricts the flexibility of the plant in an undesired way. Obviously the limit of the bottom-up approach strongly depends on the character of the plant. The control tasks of plants which are designed to support exactly one specific process (power plants, refineries..) can be structured nearly completely by the bottom-up approach while in plants designed for flexible production (chemical plants, pharmaceutical plants..) only a small part of the control task can be specified by the bottom-up approach.

2.2 Product-Related Control Tasks

From a product-oriented point of view the structuring of the control task can be started by a top-down specification of the production process. Processes can be structured by partitioning into consecutive or parallel sub-processes. The control function types that are needed to run the respective processes are described in

form of recipes. The recipes can be partitioned related to the partitioning of the process. Recipes are formulated on different abstraction levels. As described in the batch control standard [7] recipes can be specified as general recipes, site recipes, master recipes or control recipes. The most specific form is the control recipe. The control recipe contains the complete instruction sequence, which is necessary to run a specific process within a specific plant. It is possible to specify the complete control functionality in the control recipe, but this would not be a good idea. The definition of a motor logic is not a matter of a recipe specification. Therefore, the overall control solution is a combination of the bottom-up and the top-down approach.

2.3 Complete Structure

Figure 3 shows the complete structure of the control functionality as a combination of the two approaches. The concept assumes, that every production measure can be realized by an autonomous control module. This module has to be dynamically implemented as planned in the production schedule. Besides the measures to produce products, there are other measures like logistic measures, diagnostic measures and so on that can be realized by the same concept.

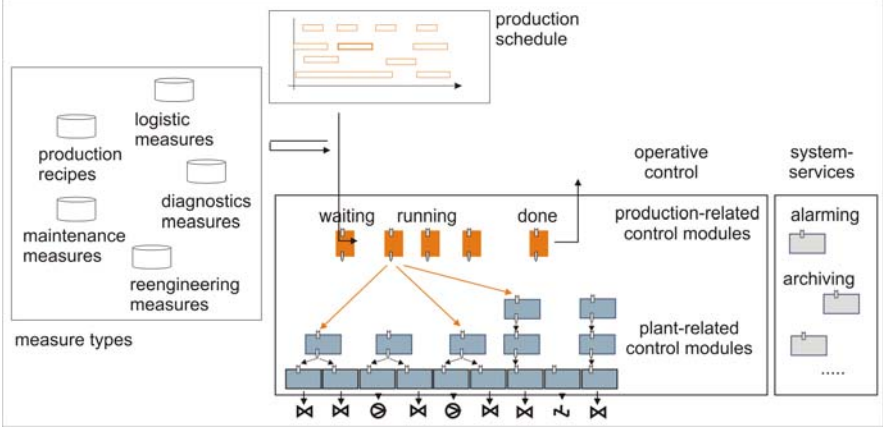


Fig. 3. The complete structure of the control functionality

Obviously *control* is the main aspect of process control engineering. But there are other aspects to be regarded, for example cross-cutting aspects like optimization, monitoring etc. and system functional aspects like HMI, alarming, archiving, etc. Currently service oriented approaches are discussed to realize these functionalities as standard blocks available for all applications of all hierarchical levels.

3 Software Architecture

The languages to describe control functions for industrial process control systems are defined in the international standard IEC61131 [6]. This is the leading standard. All important international process control systems support the languages defined in this standard. The defined languages are: Instruction List (IL), Ladder Diagram (LD), structured Text (ST), Function Block Diagram (FBD) and Sequential Function Charts (SFC). These languages serve for modeling as well as for coding. In process control engineering typically there is no distinction between modeling and coding.

In this paper only the FBD and the SFC-languages will be referenced. The main architectural principles can be explained on the base of these two languages.

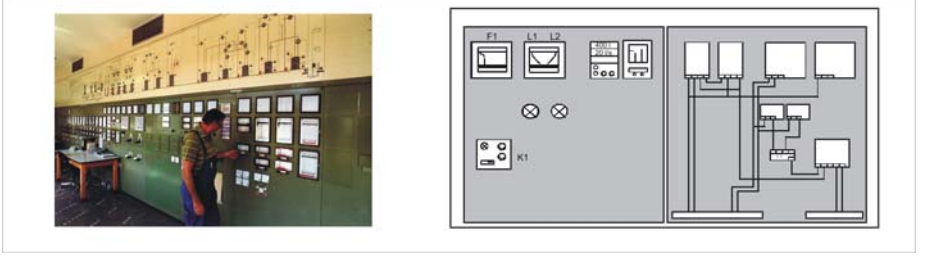


Fig. 4. The single-device technology

The FBD-language realizes the function block model. The function block model is the basic software-modeling concept in industrial process automation. The characteristic features of the function block model can be explained clearly by the old single-device technology. Figure 4 shows a central control room typical for the period around 1970. The figure on the right side shows the front and the backside of a panel. In the functional model every device corresponds to a function block and every electrical signal line to a communication connection. Some of the characteristic features can be seen directly:

- Each device executes its internal functionality independent of the other devices.
- The overall functionality is a result of the interaction of the single devices.
- The devices execute their functionality in a quasi-continuous way.

Each device calculates permanently its state values and output values. The permanent calculation can be realized by an analog circuit or by a μ-controller running a program in a fast cycle. With the adjective "quasi-continuous" it shall be expressed, that the execution cycle is very short in comparison with the technological time constants. In this case the time discreteness of the output variables can be neglected and the behavior of the device can technologically be seen to be continuous.

- The communication between the devices is realized by signal connections.
- Signal connections are separate elements (wires) and can be handled individually and independently of the devices.
- Signal connections don't affect the output value by reading

By abstracting the devices to their automation functionality we get the function block model. Figure 5 shows the automation functionality of the panel in figure 4 in form of a function block model. The one-to-one correspondence is obvious. Function blocks have inputs, internal states, outputs and a quasi-continuously executed block-method. The block-method supplies the outputs and the internal states quasi-continuously with actual values. Every block does this at every point of time, independent of the control and data flow of the application, independent of the state and the existence of other blocks and independent of existing or unexisting communication connections at the in- and outputs. Function blocks act independently and concurrently. Communication connections are separate active units. They have to assure that the values of the inputs are set to the values of the respective outputs. The signal model requires that the transmission is executed quasi-continuously and the read output value is not affected.

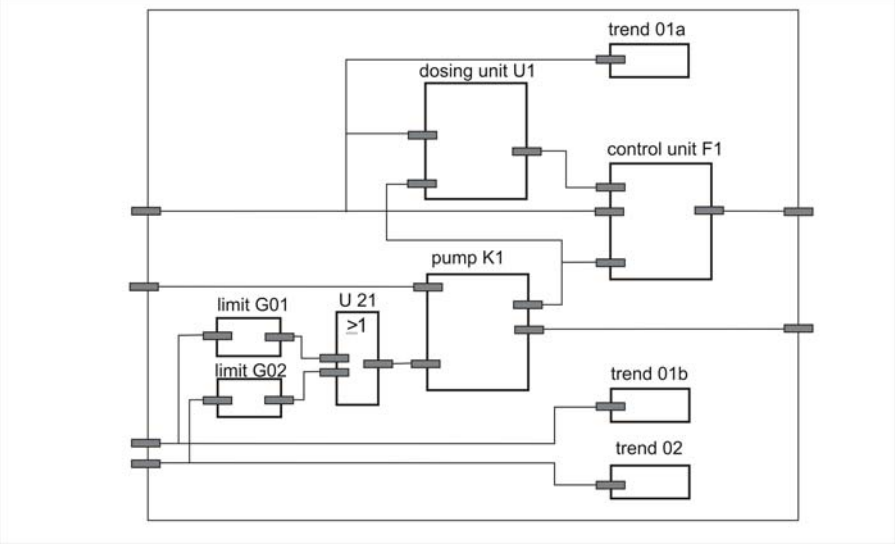


Fig. 5. Function block model

The language FBD is a graphical language to specify such a function block network. But this is not the main feature. The main feature is, that in process control systems the function block network remains explicitly visible and Δ-changeable in the execution structure of the control system. Software function

blocks are components that can be manipulated online like the devices in the single device technology. The possibility to correct and reengineer the function block network during runtime was one of the main reasons for the success of this model in the industrial practice.

If we realize the plant-related control modules in figure 3 as well as the production-related control modules by function blocks or compounds of function blocks, we get a clear programming model for the "programming in the large". The process control systems software architecture basically supports this programming model and allows a flexible dynamic change of the engineered structure even at the operating stage.

Function blocks are instances of function block types. The functionality of a block-type is either given or has to be specified explicitly by one of the automation languages IL, ST, LD, FBD or SFC. In the first "black box model" we denominate the types as *standard block types* , in the second "white box model" as *derived block types*. Figure 6 shows a valve control module. A compound of blocks as shown on the left side realizes the valve control. The compound consists of a special valve type block and a network of standard blocks to realize the role specific locking and wiring. The valve type block can be seen either as a standard block, if there is a respective type available, or has to be specified as a derived block, for example by a FBD specification as shown on the right side.

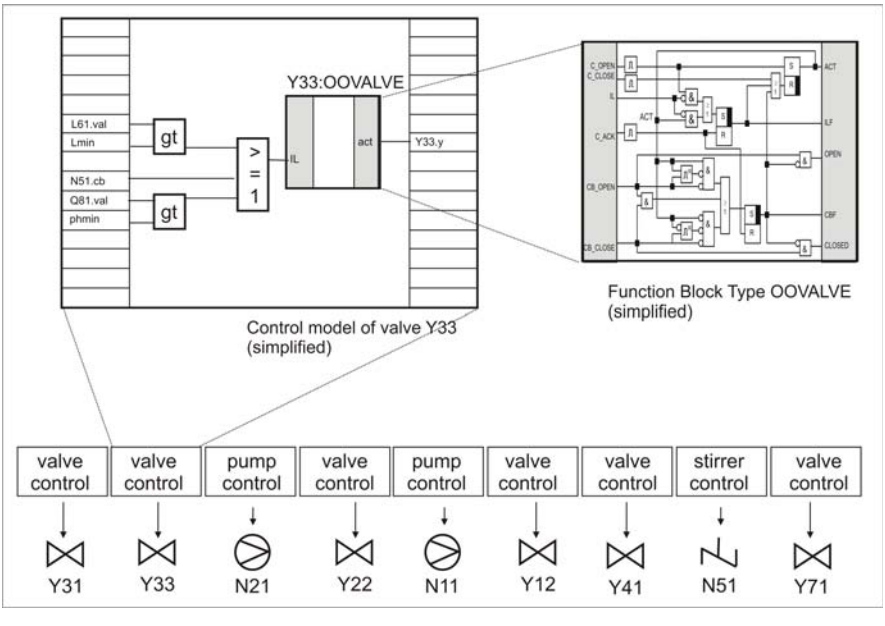


Fig. 6. Block types and compounds

For the model based engineering of production-related control modules derived blocks by SFC-specifications are of special interest. Figure 7 shows the principle of a SFC-specification. In addition to the shown structure in figure 7 the SFC syntax allows alternative and parallel branching. A SFC can be modeled as a graph with steps, transitions and action blocks as nodes. To prevent undesirable and inconsistent behavior [2] the graph has to fulfill structural restrictions.

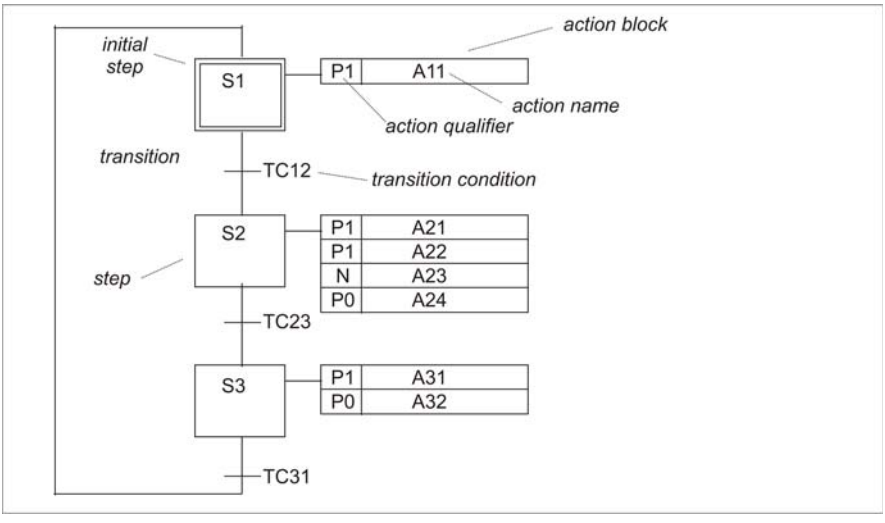


Fig. 7. Elements of a SFC

4 Control Engineering in the Operation Phase

In the next years the amount of functionality to be implemented in the process control environment will increase extensively. Requirements from the manufacturing execution level like asset management, performance monitoring, process optimization, quality assurance etc. have to be supported by the functionality of the process control level. These new functions are typically cross functions. They do not fit into the classical hierarchical architecture and the classical intuitive engineering process. New architectural principles are needed as well as new engineering processes. The already existing model driven approaches have to be improved significantly. A second trend can be stated. Figure 4.1 shows the three main phases of a plant: the process development phase, the plant-engineering phase and the operative phase. Within the process development phase the production process has to be designed and specified. To master such complex development processes model-driven approaches seem to be the right answer. The results of the IMPROVE project [12] determine the current state of the art. An engineering process is much less complicated than a development process. Within the engineering process the concrete plant design, the methods

to implement the required functionality and the technical realization are specified, as well as the procedures to operate and to maintain the plant. The control engineering process starts when the principle design of the plant is fixed. The P&I-diagram as shown in figure 1 is a typical starting point. In the past, control engineering took place primarily in the engineering phase. Modifications of the control software within the operating phase, caused for example by migrations of the control system (ca every 10 years) or by revamps of the production plant (ca every 5 years) were very seldom. But this situation is changing.

1. The migration cycles become much shorter. Most importantly, the computer components change rapidly. The software systems have to be adapted continuously to new system versions.
2. The improvement of the process performance causes a permanent need for optimization and adaptation of the control functionality.
3. Flexible plant structures, fast changing output requirements and dynamically changing optimization goals cause unplanned modifications in the control solutions.

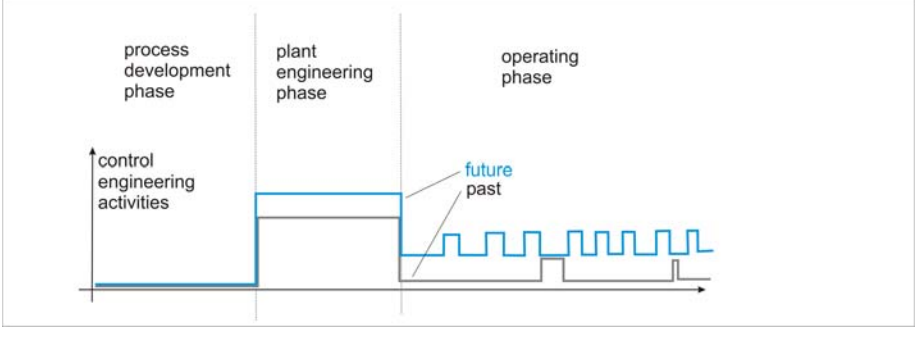


Fig. 8. Life cycle phases and control engineering activities

In the future we have to accept the control engineering process as a permanent process along the life cycle of an industrial plant. As shown in figure 8, control engineering starts with the engineering phase and will not end until the plant is removed. The implementation of a permanent control engineering process in the operating phase is a special challenge. Contrary to the situation in classical software maintenance, in the process industry the change of control software has to be realized typically without stopping the technical process and consequently the execution of the overall control system. Besides the technical problem to realize such an online changeability, it must be assured that the unconcerned parts of the software will not be affected by the change. The correctness of the concerned parts cannot be checked by a start up test, they become operative immediately.

The need for a highly formalized and systematic engineering process is obvious. In the next section current approaches to improve the engineering process are discussed.

5 Towards a Methodical, Rule Assisted and Automated Engineering

In this section some ideas will be presented on how the engineering process of industrial control software can be improved systematically. The focus is especially on the modification of existing software during the operation phase.

5.1 Modeling the Engineering Process as a Series of Single Transformation Steps

The engineering process can be seen as a series of single transformation steps. At the left side in figure 9 a function block network is shown. To change this network, for example, a new block can be created or deleted, a connection line can be created or deleted or changed and so on. All these changes can be seen as elementary transitions from a pre-step model to a post step model. Interpreting the function block model as a graph, the engineering process can be seen as a sequential graph rewriting. It has been shown that only a few rewriting types are needed to manipulate a function block system completely. Adding rewriting rules guarantees that if the pre-transformation graph is correct, the post-transformation graph is correct too. At the right side of figure 9 the rewriting rule for the implementation of a new communication connection is shown in PROGRES notation. PROGRES is a graph grammar programming environment developed by the

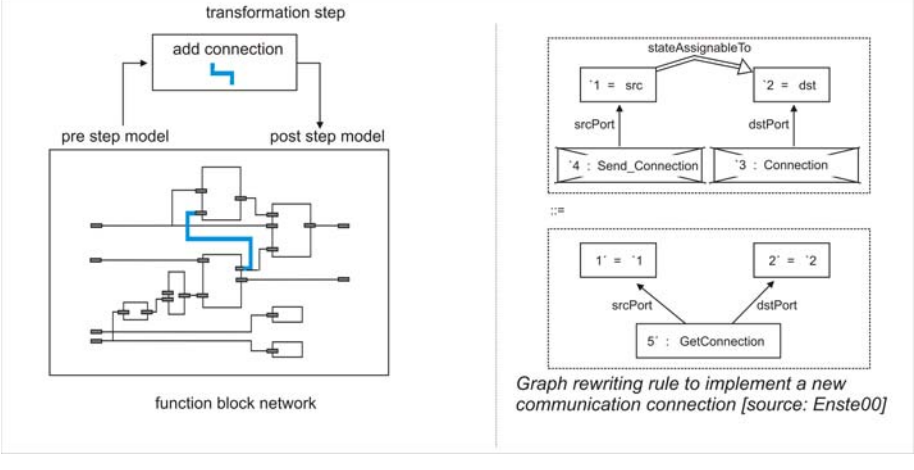


Fig. 9. Engineering as a sequential graph rewriting process

Chair of Comp. Science III at the RWTH Aachen University [11, 3]. The PROGRES environment was used by [4] to specify the function block system. Up to now the transformation of the function block network as a system of instances has been discussed, but the same procedure can be used to change the methods of derived block types declared by a SFC or FBD network. The modification of the SFC-declarations is of special interest in order to adapt the control sequences of the production-related measure types to frequently changing requirements. Currently there is no rule declaration available describing allowed and forbidden transformations of SFC-networks. Of course the commercial "recipe" systems have intrinsic rules to guarantee correct SFC-declarations but this is not sufficient. It is necessary to specify the rules explicitly and to notate them in a formal way. The formal analysis of SFC networks shown in [2] is a solid starting point.

The vision is a formally specified complete set of elementary transformation types with their respective application rules for function block networks as well as for the SFC or FBD specification of derived function block types. With this set it becomes possible to modify the control software by a sequence of correct elementary step instances.

5.2 Implementing an Engineering Service Architecture

The discussed change by elementary transformations is an operative concept. The engineering process can be realized as a sequential process. Each step can be performed separately. After each step the system is in a correct form. With this background we can implement a special service architecture to structure the engineering process. The basic system architecture is shown in figure 10. The realization of the function block model is completely hidden from the engineering process. Interaction with the model is only possible via the engineering service interface. The engineering service interface offers services to explore the model and services to realize the discussed basic transformations. For every basic transformation type a service is specified which is able to carry out the transformation with respect to the defined rules. The system divides the engineering problem into two different parts: The conception and realization of the function block management system and the design of the engineering process as a sequence of transformation service calls. These systems can be developed completely independently. They are coupled by the defined transformation services, which realize the basic transformation steps.

For the industrial process control environment the concept offers the enabling of new possibilities. The function block model can be stored in a passive data repository but it can also be stored in an active object management system and even in the online execution system. Hidden by the services, the same engineering process can be used for the classical offline engineering but also for a real-time online engineering with or without an immediate change of the active control software. Modern control systems provide object-handling services as a basic part of their runtime system. With this functionality, the realization of the engineering services for the runtime environment becomes very easy.

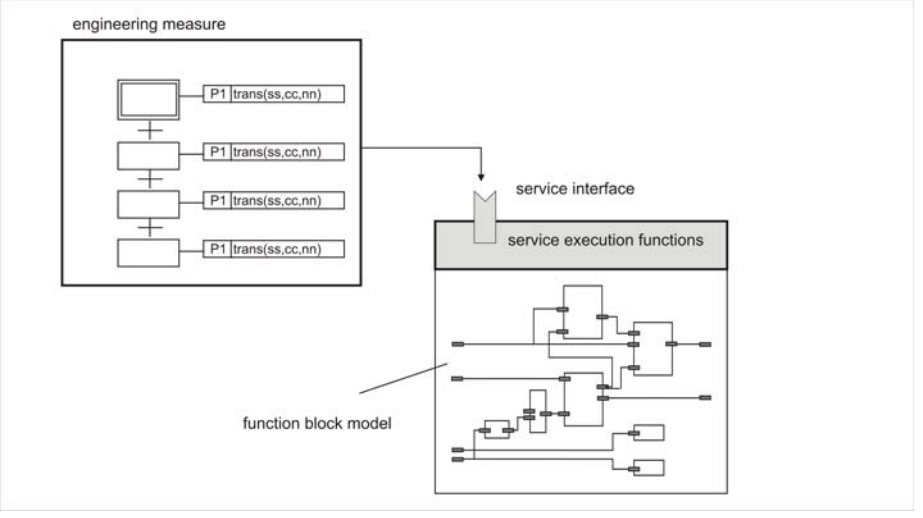


Fig. 10. The engineering service architecture

5.3 Automated and Assisted Engineering Functionality

To assist the engineer in the implementation of partially automated engineering functions is one of the hot topics in the industrial process control environment. A manifold of control functions can be specified in a rule based manner. Rules have a premise and a conclusion. Here the conclusion is always a complex transition of the function block system. This implies that the conclusion affects the function block system only. On the other hand, the premises typically need information of the structure and the characteristics of the technical plant, the processes, the configuration of the control system and so on. An important precondition to realize automated engineering functions is a formal and explicit readable form of these specifications. To get an integrated concept we define a common meta-model for all these different models and we equip all the models with the same services for exploration and data acquisition. This concept is shown in figure 11. The figure shows the models and meta-models and the possibility to change them online by engineering service requests. The architecture is hierarchical. Starting with the (meta-meta-) model ACPLT [10, 1], at the next level the (meta-) model CAEX as defined in the international standard [8] specifies a unique understanding of role-systems, components etc. Now the models split. PandIX is the base for the description of chemical plants. The FBD and SFC models are the bases for the description of the control software. Logically, the exploration services are connected to the basic object model level, but the engineering services discussed here are linked directly to the FBD, SFC-model level (they need the FBD, SFC semantics to be able to check the transformation rules).

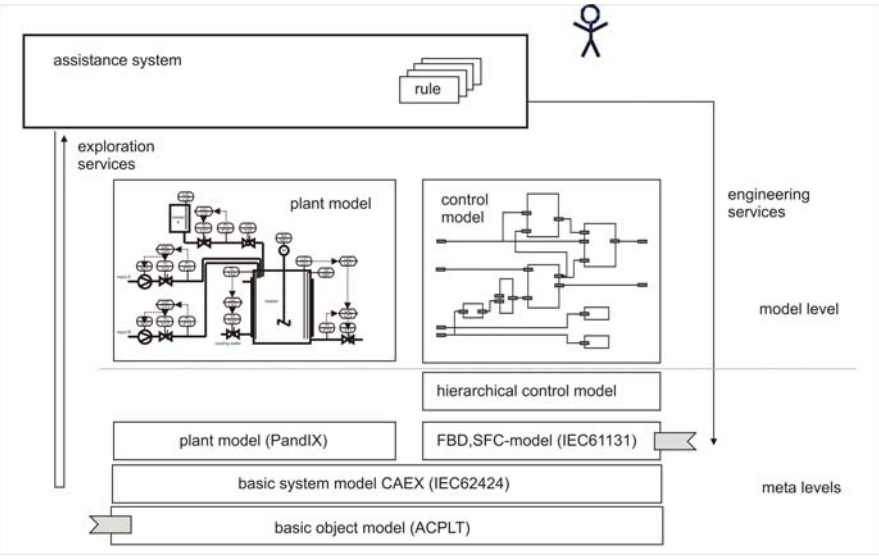


Fig. 11. Basic concept of the model landscape

5.4 Towards "Correct" Models

From model based engineering two advantages are expected: A significant increase of efficiency of the engineering process and a higher quality of the resulting control functionality. Adressing the second goal, transformation rules shall be specified to generate always "correct" models. But what is a "correct" model? This question can be split into three aspects:

- *operating system level*
On a basic "operating system level" it is required that the application must remain executable without any system failures and that the unconcerned parts of the software will not be affected by the transformation. This is a strong restriction. After every single transformation step we need a correct executable system. But this is the thinking of process control engineers and modern control systems support this feature. If for example only FBD and SFC constructs are used, the systems themselves allow only "correct" transitions and guarantee their non-interruptible execution. This is possible even for transitions in a real-time environment as all these transitions are small and can be executed very fast.

- *meta model conformence level*
On a second "meta model conformence level" the engineered models have to be kept in accordance with the design rules of the respective metamodels by the transformation process. In figure 11 for example, the "FBD/SFC model" and the "hierachical control model" are metamodels to be regarded by the control engineering process. The goal is to define general types of transformation steps - or chains of transformation steps - which guaranty that the transformed control

model remains correct in accordance with the FBD/SFC and hierachical control design rules. At the meta model conformence level it can be tolerated, that correctness is not guaranteed until the final transformation step of a chain is finished.

- *application level*
Correct operation and meta model consistency are an important base, but they do not guarantee a meaningful application functionality. To understand an application functionality, a broad and semantic complex knowledge of the goals and the nature of the application is necessary. A general rule system would be too complex. But, as shown in figure 11, most of the work is already done by defining the functionality of the plant (P&ID) and the processes (recipes). As shown in chapter 2, this information can be translated methodically into the control model. So it's not possible to assure the correctness of the application, but it is possible to assure the correct translation of the P&ID plant model information into the control model.

5.5 Rule Integration

The vision of future control concepts is an integrated decentralized real-time runtime environment with the FBD and SFC constructs as exclusive concepts. The idea is to integrate functionalities like diagnosis, engineering and so on in this concept. This seems to be possible, but there is still a problem. In any case we need rules. So we have to formulate rules within a FBD or SFC concept. Here we propose a model to describe rules by function block networks [13]. Figure 12 shows a detail of the premise test part of a rule. The shown function blocks are instances of special "rule" function block types. The "AND" type is not a simple logic type, the rule type "AND" also organizes the execution of the sub-chains to get the needed logic result at its input. A rule function block has two roles: when activated, it executes a special search or comparison question in a given context. It solves the question by own iteration and comparison methods and by activating subordinated chains to solve subordinate questions. When the answer of the question is completely ascertained, the function block stores the answer and signals its completion. The rule execution can be interrupted to fulfill real-time requirements. The last state remains stored in the involved blocks. With only a handful of "rule-block types" a powerful rule system can be established by means of usual FBD language constructs. With this concept rules become an integral part of the control software:

- rules can be engineered like other automation functions,
- rules can be engineered by other rules,
- rules can be executed in the runtime environment of the process control system.

Engineering processes can be integrated as measures in the concept shown in figure 3.

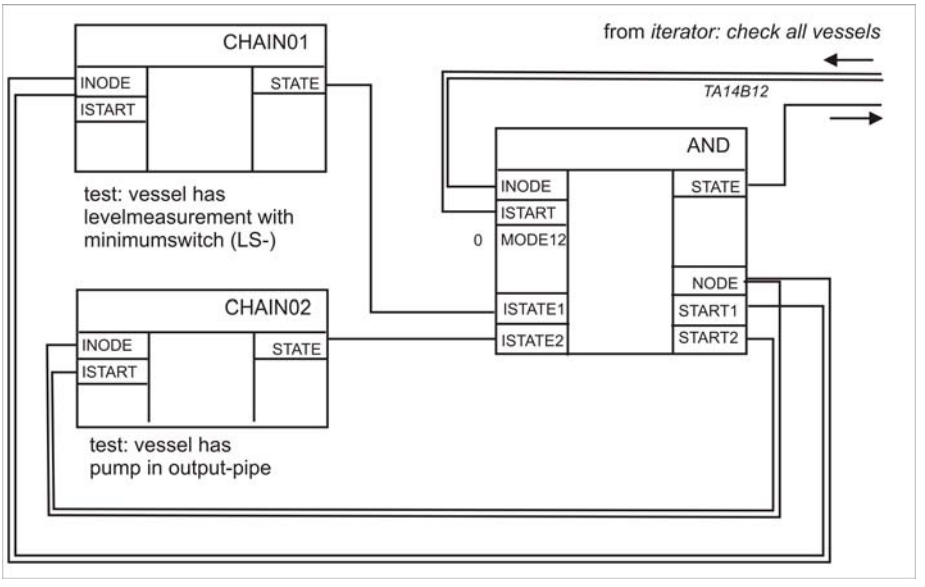


Fig. 12. Formulating rules as function-block networks

6 Summary

Industrial process control is a profitable field for model-driven approaches. There is a broad and traditional feeling for the value of models. The design problems are challenging but not too complex. Basic functions are already realized within commercial systems. The international language standards fit to model-based concepts. Engineering and maintaining industrial control software addresses a big market. The users are interested in getting more flexibility especially in the operating phase, but they are also conservative. They are interested in keeping things plain and understandable and they like their well-known language constructs like FBD or SFB. The presented concepts are fragments of an improved new concept for the process control software design. There is a realistic possibility to take a big step in this direction within the next years. But we also have to state that there are still substantial deficits in the model development and in the formal description of the specifications. Up to now, there exists no complete, verified and formally specified set of the basic transformations for FBD and SFC structures. On the other hand, the automated implementation of add-on-functions like balance supervision, asset monitoring or performance optimization by model-based transformation processes become more and more usual in the chemical industry. To encourage the users to implement such new concepts in their production plants, it is helpful to organize the software in function integrity levels. In figure 13 the function integrity model is shown. At the left side we see a two-level model. Currently this is the usual model. The control functionality is divided into two classes: safety-relevant functions and normal, non-safety

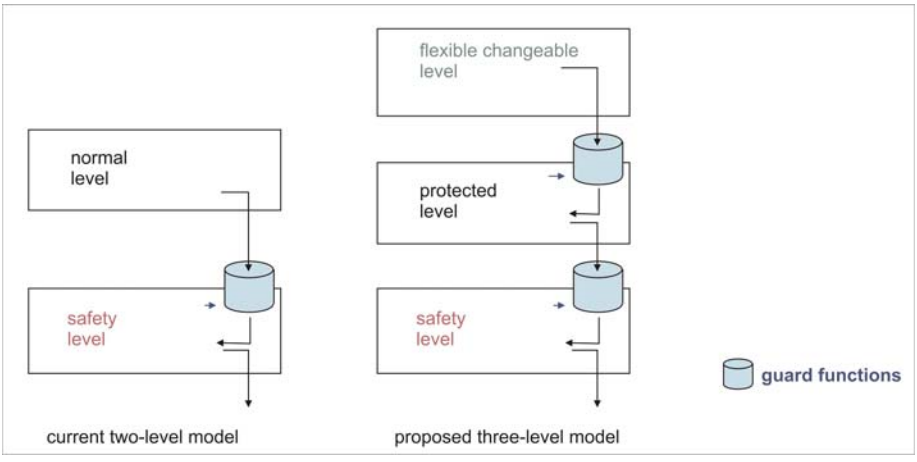


Fig. 13. Functional integrity levels

relevant functions. Guard-functions assure that changes or errors in the normal function level never affect the functionality of the underlying safety level. Safety functions have prior access to the actuators. To fulfill the discussed new flexibility requirements it seems helpful to expand the two-level model to a three level model. The safety level remains unchanged but the normal level is split into two new levels: a "protected" level and a "flexible changeable" level. The protected level contains the basic functionality that is necessary to run the plant in a normal "protected" mode, to guard the plant hardware and the product and to fulfill warranty requirements. The flexible level contains add-on functions like KPI-calculations, advanced asset diagnosis and optimization functions, or hooks up advanced control modes. The flexible and changeable level gives an ideal playground to test model-driven approaches in industrial environments.

References

[1] Albrecht, H.: On Meta-Modeling for Communication in Operational Process Control Engineering. Fortschr.Ber. VDI Reihe 8 Nr.975 VDI-Verlag Düsseldorf (2003)
[2] Bauer, N.: Formale Analyse von Sequential Function Charts. Shaker Verlag, Aachen (2004)
[3] Schürr, A.: PROGRES: A VHL-Language Based on Graph Grammars. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Grammars 1990. LNCS, vol. 532, pp. 641–659. Springer, Heidelberg (1991); also: Technical Report AIB 90-16, RWTH Aachen, Germany (1990)
[4] Enste, U., Kneissl, M.: Development of standardized process control software - avoiding bugs by using graph grammars. In: 3rd Mathmod Vienna, IMACS Symposium on Mathematical Modeling, Wien S., pp. 381–384 (2000)
[5] Enste, U.: Generische Entwurfsmuster in der Funktionsbausteintechnik und deren Anwendung in der operativen Prozessführung. VDI Fortschr.Ber., Reihe 8, Nr. 884. VDI-Verlag Düsseldorf (2001)

[6] IEC61131-3: Programmable controllers - Part 3: Programming languages. Publication IEC
[7] IEC61512: Batch control. Publication IEC
[8] IEC62424: Representation of process control engineering-requests in P&I diagrams and data exchange between P&ID tools and PCE-CAE tools. Publication IEC
[9] IEC62264-1: Enterprise-control system integration. Part 1: Models and terminology. Publication IEC (2003)
[10] Meyer, D.: Objektverwaltungskonzepte für die operative Prozessleittechnik. Fortschr.Ber. VDI Reihe 8 Nr.940 VDI-Verlag Düsseldorf (2002)
[11] Nagl, M. (ed.): Proc. WG 1989 Workshop on Graph-Theoretic Concepts in Computer Science. LNCS, vol. 411. Springer, Heidelberg (1989)
[12] Nagl, M., Marquardt, W. (eds.): Collaborative and Distributed Chemical Engineering. LNCS, vol. 4970. Springer, Heidelberg (2008)
[13] Schmitz, S., Epple, U.: Automatisiertes Engineering leittechnischer Funktionen durch integrierte Regeln. Entwurf komplexer Automatisierungssysteme (EKA) 2008: Beschreibungsmittel, Methoden, Werkzeuge und Anwendungen; 10. Fachtagung, S. 241–S. 252, April 15-17 (2008)

Author Index