Iris Reinhartz-Berger · Arnon Sturm
Tony Clark · Sholom Cohen
Jorn Bettin *Editors*

Domain Engineering

Product Lines, Languages, and Conceptual Models

Domain Engineering

Iris Reinhartz-Berger • Arnon Sturm •
Tony Clark • Sholom Cohen • Jorn Bettin
Editors

Domain Engineering

Product Lines, Languages, and Conceptual Models

Editors
Iris Reinhartz-Berger
Dept. Information Systems
University of Haifa
Haifa
Israel

Arnon Sturm
Dept. Information Systems Engineering
Ben-Gurion University of the Negev
Beer-Sheva
Israel

Tony Clark
Business Information Systems
Middlesex University
London
United Kingdom

Sholom Cohen
Software Engineering Institute
Pittsburgh
Pennsylvania
USA

Jorn Bettin
S23M
Melbourne
Australia

ISBN 978-3-642-36653-6 ISBN 978-3-642-36654-3 (eBook)
DOI 10.1007/978-3-642-36654-3
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013942395

ACM Computing Classification: D.2, I.6

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface: Introduction to Domain Engineering: Product Lines, Languages, and Conceptual Models

A domain is an area of knowledge that uses common concepts for describing phenomena, requirements, problems, capabilities, and solutions. A domain is usually associated with well-defined or partially defined terminology. This terminology refers to the basic concepts in that domain, their definitions (i.e., their semantic meanings), and their relationships. It may also refer to behaviors that are desired, forbidden, or perceived within the domain. *Domain engineering* is a set of activities that aim to develop, maintain, and manage the creation and evolution of domains.

Domain engineering has become of special interest to the information systems and software engineering communities for several reasons. These reasons include, in particular, the need to maintain and use existing knowledge, the need to manage increasing requirements for variability of information and software systems, and the need to obtain, formalize, and share expertise in different, evolving domains.

Domain engineering as a discipline has practical significance as it can provide methods and techniques that may help reduce time-to-market, development cost, and projects risks, on the one hand, and help improve product quality and performance on a consistent basis, on the other hand. It is used, researched, and studied in various fields, predominantly: software product line engineering (SPLE), domain-specific language engineering (DSLE), and conceptual modeling.

This book presents a collection of state-of-the-art research studies in the domain engineering field. About half of the chapters in this collection originated from a series of workshops, named domain engineering, which were associated with the Conference of Advanced Information Systems Engineering (CAiSE) during the years 2009–2011 and with the international conference on conceptual modeling (also known as the ER conference) in 2010. The authors of the other chapters were personally invited to contribute to this book. The chapters are organized in three parts. The first part includes research studies that deal with domain engineering in SPLE. The second part refers to domain engineering as a research topic within the field of DSLE. Finally, the third part presents research studies that deal with domain engineering within the field of conceptual modeling.

Part I: Software Product Line Engineering

A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [1, 8]. While reuse has always made sense as a means to take advantage of the commonality across systems, most reuse strategies fail to have any real technical or economic impact. SPLE is a discipline that addresses technical and economic benefit and achieves strategic reuse of software across the product line through: (1) capturing common features and factoring the variations across the domain or domains of a product line; (2) developing core assets used in constructing the systems of the product line; (3) promulgating and enforcing a prescribed way for building software product line assets and systems; and (4) evolving both core assets and products in the product line to sustain their applicability.

Although SPLE has been a recognized discipline within the software engineering community for two decades, the practice of SPLE in industry still faces significant challenges in achieving strategic reuse. Challenges are seen in each of the four areas mentioned above. Specific examples include: (1) capturing commonality—modeling and representation approaches, tools, and analysis for variation and variation management; (2) developing assets—architecture design approaches including real-time embedded, design patterns, automatic generation of software, and design approaches including aspect-orientation; and (3) building software—implementation of software assets for reuse, composition techniques, and use of domain-specific languages (DSLs) for software construction. In addition, there is a need to deal with the evolution of the core assets and the product line systems and a need for specific tools to assist the coevolution of product line core assets and dependent systems.

In the field of SPLE, domain engineering deals with specifying, designing, implementing, and managing reusable assets, such as specification sets, patterns, and components, that may be suitable, after customization, adaptation, or even extension, to families of software products. The focus of domain engineering in this field is on conducting commonality and variability analysis and representing the results of this analysis in a comprehensible way. Commonly, feature-oriented methods and UML profiles are used for this purpose.

The chapters in this part of the book deal with application of domain engineering to address some of the aforementioned challenges. Two chapters deal with domain engineering to capture commonality and manage variation across a software product line:

- "Separating concerns in feature models: Retrospective and support for multi-views" by Mathieu Acher, Arnaud Hubaux, Patrick Heymans, Thein Than Tun, Philippe Lahire, and Philippe Collet looks at managing common features and their variants in software product lines with thousands of features. This chapter describes the separation of concerns, that can be applied to partition the feature space.

- "A survey of feature location techniques" by Julia Rubin and Marsha Chechik examines over 20 techniques that offer automated or semi-automated approaches to isolate features from existing software. This chapter provides a description of the overarching technology for isolating features for software code and analyzes the potential that each of the 20+ techniques offers. The chapter also provides guidance in selecting the appropriate technique.

One chapter deals with architecture and design for software product lines, specifically those software product lines that are real-time (RT) embedded:

- "Modeling real-time design patterns with the UML-RTDP profile" by Saoussen Rekhis, Nadia Bouassida, Rafik Bouaziz, Claude Duvallet, and Bruno Sadeg applies domain engineering to design RT patterns for capturing commonality and managing variation across the software product line. This chapter introduces UML-based models that represent the static and dynamic patterns of a software product line architecture for RT systems. It describes the application of these models in an example of an RT control system.

The two remaining chapters in this part apply domain engineering to develop techniques for building core assets and systems in the software product line:

- "When aspect-orientation meets software product line engineering" by Iris Reinhartz-Berger discusses an approach that melds aspect-oriented and SPLE methods through domain engineering. The approach uses the Application-based DOmain Modeling (ADOM) method to support families of aspects and weave them to families of software products. Reuse is enhanced through the three levels addressed by ADOM: language, domain, and application.
- "Utilizing application frameworks: a domain engineering approach" by Arnon Sturm and Oded Kramer also uses ADOM. In this chapter, the modeling approach supports specification and use of frameworks as a construct to support reuse across the software product line. ADOM also contributes to domain-specific languages to reduce the development effort and increase their reusability and code quality.

Part II: Domain-Specific Language Engineering

DSLs are specification or programming languages tailored to specific domains [6, 7]. These languages are developed in a domain engineering process and are later used to develop and maintain solutions (i.e., software systems) in the specific domains. The focus on a specific domain is achieved by abstracting from general programming language implementation details such as variable locations and control structures. The resulting DSL features correspond to domain elements and are often referred to as declarative (as opposed to imperative) because they focus on expressing desirable states of the problem domain rather than computations in the solution domain. The benefits of adopting DSLs include increased productivity,

improved quality, reuse of experts' knowledge, and, perhaps most importantly, better maintainability [6, 7].

Conceptually, a DSL is a formal language that is expected to be understood by domain experts and that can be interpreted by domain-specific software tools to produce lower level specifications. Practically, a DSL is a preferably small language that focuses on a particular aspect of software systems (e.g., [3]) and that is used by domain-specific software tools to generate code in a general purpose programming language or other lower level formal specification languages, e.g., [7]. Examples of domain-specific programming approaches are elaborated in [4, 5].

Domain-specific language engineering (DSLE) is concerned with methods and tools for specifying and utilizing such languages. This includes the identification of relevant language concepts and their relationships, the determination of the most appropriate level of abstraction for the envisaged users of the language, and the specification of all required transformations.

Although the practice of DSLE has evolved considerably in the last two decades, a number of challenges remain. There is a need to study the ways in which people use DSLs and to what extent. In line with the increasing popularity of DSLs, there is a need to evaluate the ways in which DSLs are composed, to examine maintainability and interoperability, and to devise mechanisms that enable end users to extend DSLs. The trade-offs between using general purpose languages and DSLs also merit further discussion. From an engineering perspective, there is a need to explore how DSL elements can be reused, which types of transformation are required, what best practices can be distilled from detailed case studies, how to define and evolve the semantics of a DSL, and how to evaluate the design and implementation of a DSL.

In this part of the book, we have gathered five chapters that address some of the challenges mentioned above.

The first chapter discusses the notion of domain-specific languages:

- "Domain-specific modeling languages—requirement analysis and design guidelines" by Ulrich Frank attempts to provide instructions for developing a domain-specific modeling language. In particular, this chapter introduces a set of guidelines, which consist of requirements for the meta-modeling language, as well as a detailed process description of the stages to devise a new DSL.

The following two chapters discuss the design process of domain-specific languages. Designing a DSL involves the creation of a new formal language, and therefore it is important to investigate the emergence of new languages as well as their engineering:

- "DSLs and standardization: Friends or foes?" by Øystein Haugen argues that creating a good language requires knowledge not only of the domain but also of the language design process. This chapter discusses the tension between DSLs and standardization efforts, demonstrates how DSLs can benefit from standardization, and provides a comprehensive example of language evolution and standardization.

- "Domain engineering for software tools" by Tony Clark and Balbir Barn proposes a language-driven approach that elaborates the notion of domain-specific tool chains and related tool interoperability challenges. The approach presented in the chapter views domains as languages and emphasizes the need for modularity, in particular the need for modular composition of domains and tool chains. The suggested approach for tool design involves a model of the semantic domain, a model of the abstract syntax, a model of the concrete syntax, as well as a model of the relationships between the semantic domain and the abstract syntax.

A key motivation for developing one or more DSLs for the same domain is the desire to capture all the meta-data that is needed to automate the production of detailed artifacts (such as code) from the abstract concepts supported by the DSLs. A common way of producing derived artifacts is through model transformation. Although a large number of model transformation languages have been developed, there are only few heuristics for engineering model transformation languages. The fourth chapter in this part tackles this issue:

- "Modeling a model transformation language" by Eugene Syriani, Jeff Gray, and Hans Vangheluwe introduces a technique for developing model transformation languages that refers to each language as a DSL and that includes a model of all domain concepts at the appropriate level of abstraction.

As developing a DSL is a complex task that involves stakeholders from different disciplines, a cooperative environment that supports cross-disciplinary collaboration is required. The fifth and last chapter in this part addresses this challenge:

- "A Reconciliation framework to support cooperative work with DSM" by Amanuel Alemayehu Koshima, Vincent Englebert, and Philippe Thiran proposes a communication framework that links the changes made by the language engineers and their effects on DSL users. This framework is concerned with the effects of language evolution and the propagation of changes in tool chains and across the stakeholders and the language user community.

Part III: Conceptual Modeling

Before any system can be collaboratively developed, used, and maintained, it is necessary to study and understand the domain of discourse. This is commonly done by developing a conceptual model. The main purposes of conceptual models are: (1) supporting communications between different types of stakeholders and especially between developers and users; (2) helping analysts understand the domain of interest, its terminology, and rules; (3) providing input for the next development phases, namely top level and detailed design; and (4) documenting the requirements that originate from the real world for maintenance purposes and future reference.

The process of building conceptual models, conceptual modeling, involves developing and maintaining representations of selected phenomena in the application domain [12]. These representations, the conceptual models, are usually developed during the requirements analysis phase of software or information systems development. Such models aim to capture the essential features of systems in terms of the different categories of entity, their properties, relationships, and their meaning. They are used for representing both structural and dynamic phenomena, usually in a graphic way. Once a conceptual model is constructed and agreed on, it forms a foundational basis for subsequent engineering activities.

Although research in conceptual modeling has existed for many years, its boundaries are quite vague. In particular, conceptual modeling has significant overlap with the field of knowledge engineering [9]. Many of the features of modern notations for conceptual modeling can be traced to examples in both early system design notations and knowledge representation notations, such as conceptual structures [10]. Conceptual modeling also has a strong relationship to ontologies [11], and the question whether conceptual models and ontologies are alternatives of each other is open. Clearly, Conceptual modeling is also related to model-driven architecture (MDA), which has become a significant research topic in recent years. MDA promotes the idea that systems should be modeled at a high level of abstraction and then systems are partially or completely generated from the models.

In light of technology improvements, many challenges that concern domain engineering in the context of conceptual modeling arise. In particular, how can the real world be modeled to better support the development, implementation, use, and maintenance of systems? [1] Conceptual modeling applies to many different application domains which raises the question of how to support the representational needs of each domain. Should there be a single universal language for conceptual modeling or several different languages? Do methods that apply to one type of application (finance for example) also apply in another (for example an embedded system)? The representational issue is often addressed using meta-techniques that allow the conceptual modeler to use a standard notation to design a bespoke notation that is used to express the conceptual model. While the best meta-technology is an open question, UML provides profiles that allow the UML standard to be tailored in a number of ways to support new concepts in terms of abstract modeling elements and the ways they are represented on diagrams. Also, following the evolvement of the MDA approach, it is interesting to examine how conceptual models fit into various MDA technologies and processes. Finally, the management of conceptual models is also a challenge, especially those that involve meta-technologies [2]. In addition to the usual problems related to distributed multi-person development, a conceptual model written using a notation that has been specifically defined for this purpose requires care when the meta-model is evolved, otherwise the conceptual model becomes meaningless.

The chapters in this part of the book address some of the challenges mentioned above. The first chapter in this part suggests using domain engineering for formalizing the knowledge of domain experts.

- "Model oriented domain analysis and engineering" by Jorn Bettin presents a model-oriented domain analysis and engineering methodology. This methodology, whose roots are in both SPLE and conceptual modeling, can be used to uncover and formalize the knowledge that is inherent in any software-intensive business or any scientific discipline.

The second chapter analyzes the relationships between different abstraction levels of modeling in order to support the definition of domain-specific modeling languages.

- "Multi-level meta-modeling to underpin the abstract and concrete syntax for domain-specific modelling languages" by Brian Henderson-Sellers and Cesar Gonzalez-Perez discusses the relationships between models, meta-models, modeling languages, and ontologies. They further provide a theoretical foundation for the construction of domain-specific modeling languages, exemplifying this foundation on two languages: ISO/IEC 24744 that can be used to define software-intensive development methods and FAML that can be used for the specification of agent-oriented software systems.

The third chapter discusses an ontology-based framework for evaluating and designing conceptual modeling languages.

- "Ontology-based evaluation and design of visual conceptual modeling languages" by Giancarlo Guizzardi addresses another methodological issue and focuses on the evaluation of the suitability of a language to model a set of real-world phenomena in a given domain. In the proposed approach, the suitability can be systematically evaluated by comparing the level of homomorphism between a concrete representation of the worldview underlying the language and an explicit and formal representation of a conceptualization of that domain (represented as a reference ontology).

The fourth chapter addresses the challenge of managing conceptual models in distributed multi-person development.

- "Automating the interoperability of conceptual models in specific development domains" by Oscar Pastor, Giovanni Giachetti, Beatriz Marín, and Francisco Valverde discusses the model management, interoperability, and reuse. In particular, it discusses the problems related to conceptual interoperability across applications in a domain. This chapter introduces a framework for describing levels of conceptual interoperability and the challenges that must be overcome to achieve the various levels and then outlines a process for achieving and automating interoperability through the integration of modeling languages.

As mentioned before, MDA promotes the idea that systems should be modeled at a high level of abstraction and then systems are partially or completely generated from the models. The benefits that are claimed for this approach are that it shields the developer from constantly changing technology platforms, increases quality, and

makes change easier to manage. The last chapter exemplifies this notion for the domain of geographic databases.

- "Domain and model-driven geographic database design" by Jugurta Lisboa-Filho, Filipe Ribeiro Nalon, Douglas Alves Peixoto, Gustavo Breder Sampaio, and Karla Albuquerque de Vasconcelos Borges describes the use of the MDA approach in the design of databases in the geographical domain. In particular, a UML Profile, called GeoProfile, is proposed and is aligned with international standards of the ISO 191xx series. This chapter also shows that with the automatic transformation of models it is possible to achieve the generation of scripts for spatial databases from a conceptual data schema in a high level of abstraction.

Concluding and Further Remarks

As elaborated above, domain engineering is closely related to several fields, primarily SPLE, DSLE, and conceptual modeling. This book provides a collection of research studies that are related to these three fields. The fields promote domain engineering differently; however, they do have significant overlap. In particular, some of the studies could pertain to more than one field. Therefore, we confirmed our classification with the authors in these cases. Moreover, as the studies are very diverse, they address a variety of important topics related to domain engineering, stressing the importance of this field, providing solutions, and further clarifying existing related challenges.

We believe that the chapters in this book are of interest to researchers, practitioners, and students of domain engineering in general and of the fields of SPLE, DSLE, and conceptual modeling in particular. Furthermore, given the exponential growth of data on the Web and the growth of the "Internet of Things," Domain Engineering research may be relevant to other disciplines as well. For example, the emergence of deep chains of Web services highlights that the service concept is relative. A set of Web services developed and operated by one organization can be utilized as part of a platform by another organization. This calls for appropriate conceptual models as well as for DSLs that together facilitate the design of service-oriented architectures. Furthermore, as services may be used in different contexts and hence require different configurations, inspiration to their design as families of services can be taken from the field of SPLE.

Another new opportunity for research is related to the Big Data domain. Big Data is characterized by the "three Vs": **v**olume, **v**ariety, and **v**elocity (rate of change). The ability to process such data depends on understanding and manipulating the information. Conceptual models can be of great help since they capture the semantics of the information which is important for making matches in the presence of incomplete and noisy input. Furthermore, meta-processing, such as dependency analysis, model transformations, model merge, and slicing, can be used to address

multiple data sources. DSLs can be used to develop languages that express domain-specific data patterns and SPLE can be utilized to help address the need to modify the patterns on a regular basis.

Lastly, we would like to thank the authors for their contribution to this book and to wish the readers enjoyable and fruitful reading.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, 3rd edn. Addison-Wesley Professional, Boston (2001)
2. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Proceedings of the 2nd International Workshop on Model Comparison in Practice (IWMCP '11), pp. 30–38 (2011)
3. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, Boston (2010)
4. Freemanand, S., Pryce, N.: Evolving an embedded domain-specific language in Java. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, pp. 855–865 (2006)
5. Kabanov, J., Raudjärv, R.: Embedded type safe domain specific languages for Java. In: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, pp. 189–197 (2008)
6. Kelly, S., Tolvanen, J-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, Hoboken (2008)
7. Mernik, M., Heering, J., Sloane, A. M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
8. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
9. Schreiber, G. T., Akkermans, H.: Knowledge Engineering and Management: The Common KADS Methodology. MIT Press, Cambridge (2000)
10. Sowa, J.: Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley Longman Publishing, Boston (1984)
11. Sugumarana, V., Storeyb, V.C.: Ontologies for conceptual modeling: their creation, use, and management. Data Knowl. Eng. **42**(3), 251–271 (2002)
12. Wand, Y., Weber, R.: Research commentary: information systems and conceptual modeling—a research agenda. Inf. Syst. Res. **13**(4), 363–376 (2002)

Haifa, Israel — Iris Reinhartz-Berger
Beer-Sheva, Israel — Arnon Sturm
Hendon, London — Tony Clark
Pittsburgh, PA — Sholom Cohen
Mordialloc, Australia — Jorn Bettin

Contents

Part I Software Product Line Engineering (SPLE)

Separating Concerns in Feature Models: Retrospective and Support for Multi-Views ... 3
Arnaud Hubaux, Mathieu Acher, Thein Than Tun, Patrick Heymans, Philippe Collet, and Philippe Lahire

A Survey of Feature Location Techniques ... 29
Julia Rubin and Marsha Chechik

Modeling Real-Time Design Patterns with the UML-RTDP Profile ... 59
Saoussen Rekhis, Nadia Bouassida, Rafik Bouaziz, Claude Duvallet, and Bruno Sadeg

When Aspect-Orientation Meets Software Product Line Engineering ... 83
Iris Reinhartz-Berger

Utilizing Application Frameworks: A Domain Engineering Approach ... 113
Arnon Sturm and Oded Kramer

Part II Domain-Specific Language Engineering (DSLE)

Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines ... 133
Ulrich Frank

Domain-Specific Languages and Standardization: Friends or Foes? ... 159
Øystein Haugen

Domain Engineering for Software Tools ... 187
Tony Clark and Balbir S. Barn

Modeling a Model Transformation Language ... 211
Eugene Syriani, Jeff Gray, and Hans Vangheluwe

A Reconciliation Framework to Support Cooperative Work with DSM 239
Amanuel Alemayehu Koshima, Vincent Englebert, and Philippe Thiran

Part III Conceptual Modeling

Model Oriented Domain Analysis and Engineering 263
Jorn Bettin

Multi-Level Meta-Modelling to Underpin the Abstract and Concrete Syntax for Domain-Specific Modelling Languages 291
Brian Henderson-Sellers and Cesar Gonzalez-Perez

Ontology-Based Evaluation and Design of Visual Conceptual Modeling Languages 317
Giancarlo Guizzardi

Automating the Interoperability of Conceptual Models in Specific Development Domains 349
Oscar Pastor, Giovanni Giachetti, Beatriz Marín, and Francisco Valverde

Domain and Model Driven Geographic Database Design 375
Jugurta Lisboa-Filho, Filipe Ribeiro Nalon, Douglas Alves Peixoto, Gustavo Breder Sampaio, and Karla Albuquerque de Vasconcelos Borges

Index 401

Part I
Software Product Line Engineering (SPLE)

Separating Concerns in Feature Models: Retrospective and Support for Multi-Views

Arnaud Hubaux, Mathieu Acher, Thein Than Tun, Patrick Heymans, Philippe Collet, and Philippe Lahire

Abstract Feature models (FMs) are a popular formalism to describe the commonality and variability of a set of assets in a software product line (SPL). SPLs usually involve large and complex FMs that describe thousands of features whose legal combinations are governed by many and often complex rules. The size and complexity of these models is partly explained by the large number of concerns considered by SPL practitioners when managing and configuring FMs. In this chapter, we first survey concerns and their separation in FMs, highlighting the need for more modular and scalable techniques. We then revisit the concept of view as a simplified representation of an FM. We finally describe a set of techniques to specify, visualise and verify the coverage of a set of views. These techniques are implemented in complementary tools providing practical support for feature-based configuration and large-scale management of FMs.

Keywords Configuration • Feature model • Model management • Separation of concerns • Slicing • Software product lines • Variability • Views

A. Hubaux (✉) · P. Heymans
PReCISE Research Centre, University of Namur, Belgium
e-mail: ahu@info.fundp.ac.be; phe@info.fundp.ac.be

M. Acher
University of Rennes 1, Irisa and INRIA, France

PReCISE Research Centre, University of Namur, Belgium
e-mail: mathieu.acher@irisa.fr

T.T. Tun
Department of Computing, The Open University, UK
e-mail: t.t.tun@open.ac.uk

P. Collet · P. Lahire
Université de Nice Sophia Antipolis - I3S (CNRS UMR 6070), France
e-mail: collet@i3s.unice.fr; lahire@i3s.unice.fr

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_1, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

In many application domains, such as avionics, telecommunications or automotive, organisations build software-intensive systems that are similar to each other. Rather than re-developing each system from scratch, these organisations reuse common software artefacts on a large scale.

The paradigm of *software product line* (SPL) engineering has emerged to support the modelling and development of software system families rather than individual systems. It aims at efficiently producing and maintaining multiple similar software products. This is analogous to the automotive industry, where the focus is on creating a single production line, out of which many customised but similar variations of a car model are produced. The key principle is to institutionalise reuse throughout the development process to obtain economies of scale and scope [53]. To achieve reuse, SPL engineering is usually separated in two complementary phases: *domain engineering* and *application engineering*. Domain engineering starts with *domain analysis*, which documents commonality (i.e., common parts of products) and variability (i.e., differences between products). Reusable assets that satisfy these descriptions are then modelled and implemented. During application engineering, the required assets are selected and possibly extended to derive, as quickly and efficiently as possible, an appropriate product. To be successful, the investments required to develop the reusable artefacts during domain engineering must be outweighed by the benefits of deriving the individual products during application engineering [25]. Domain analysis is therefore a crucial phase.

To date, feature modelling has been recognised as one of the most popular domain analysis techniques. Introduced in the 1990s and now widely adopted, *feature models* (FMs) are a simple formalism whose main purpose is to document variability in terms of *features*, i.e., domain abstractions or functionalities relevant to stakeholders [19]. The main concepts of the language are features and relationships between features. FMs have been given a formal semantics [59] which opened the way for safe and efficient automation of various, otherwise error-prone and tedious tasks such as consistency checking, FM merging and product counting. A repertoire of such automations can be found in [12].

A particular type of automation is *feature-based configuration* (FBC). FBC is an interactive process during which one or more stakeholders select and discard features to build a specific product. Traditionally, FBC systems support FM modelling, analysis and configuration. Currently, FBC techniques and tools facilitate the work of stakeholders in various ways, including: decision verification and propagation [22, 38, 47]; auto-completion [21, 38]; scheduling of configuration tasks [20, 23, 35] and alternative representations of FMs [15, 17].

FMs, and therefore FBC, are becoming increasingly large and complex. FMs are not only used to describe variability in software designs but also variability in different contexts, at different times in the development, and in different parts of the system [20, 32, 41, 50, 57]. Consequently, the list of *concerns* that may be considered in an FM is very comprehensive [9, 34, 64] ranging from hardware description [41],

organizational structure [57], business to implementation details [50]. Concerns are related in numerous ways and there can be thousands of features whose legal combinations are governed by many and often complex rules.

Furthermore, it has been observed that maintaining a single large FM for the entire system may not be feasible [26, 54]. With FMs being increasingly complex, describing various concerns of an SPL and handled by several stakeholders (or even different organizations), managing them with a large number of related features is intuitively a problem of *separation of concerns* (SoC) [11, 61]. The sought benefits are indeed similar to the ones of software engineering disciplines, i.e., reduced complexity, improved reusability and simpler evolution [61]. A possible way to achieve SoC is then to rely on views, i.e., simplified representations of an FM tailored for a specific stakeholder, role, or task [36]. Views facilitate the decision-making process in that they only focus on those parts of the FM that are relevant for a given concern.

In this chapter, our goal is to give a clear overview of existing approaches in the field and state-of-the-art techniques for separating concerns in FMs. The intended audience is domain analysts or SPL practitioners working with FMs with an interest for FBC. In the first part of this chapter, we present a review of SoC in FMs. SoC has spawned much research on FM separation, composition and analysis. Here, we reuse some material presented in [36] and focus on concerns and their *separation* in FMs and FBC. We highlight the need for more modular and scalable techniques and revisit the concept of views. In the second part of this chapter, we focus on the creation of consistent views and the generation of alternative visualisations for FBC. We present and compare two techniques to synthesise visualisations of an FM. We also report on the progress made in developing tool support for SoC and multi-view FBC.

The rest of this chapter is organised as follows. Section 2 re-examines the basics of FMs and introduces our working example. Section 3 presents the general problem of SoC in FM and reviews existing works in the field. Section 4 describes a set of SoC techniques to specify, automatically generate and check multiple views. Section 5 presents the tools supporting it.

2 Background

2.1 Feature-Based Configuration

Schobbens et al. [59] defined a generic formal semantics for a wide range of FM dialects. In essence, an FM d is a hierarchy of features (typically a tree) topped by a root feature. An FM is informally defined as follows.

Definition 1 (FM (adapted from [59])). An FM d is a tuple $(N, r, \lambda, DE, \Phi)$ where N denotes the set of features. $r \in N$ the root of the feature tree. $\lambda : N \to \mathbb{N}\times\mathbb{N}$

Table 1 FM decomposition operators

Concrete syntax	f g h	f g h	f g h	f ⟨i..j⟩ g h	f g h
Decomposition operator	and: $\wedge$	or: $\vee$	xor: $\oplus$	Generalized cardinality	Optional
Cardinality	<n..n>	<1..n>	<1..1>	<i..j>	<0..1>

denotes the cardinality $\langle i..j \rangle$ attached to a feature, where i (resp. j) is the minimum (resp. maximum) number of children (i.e., features at the level below) required in a product (aka configuration). For convenience, common cardinalities are denoted by Boolean operators, as shown in Table 1. $DE \subseteq N \times N$ denotes the decomposition edges, i.e., the parent–child relationship. Additional constraints that crosscut the tree ($\Phi \in \mathbb{B}(N)$) can also be added and are defined, without loss of generality, as a conjunction of Boolean formulae.

The semantics of an FM is the set of configurations (also called products), denoted $[\![d]\!]$, where each configuration is a combination of selected features. The full syntax and semantics as well as benefits, limitations and applications of FMs are extensively discussed elsewhere [12, 59].

FBC tools use FMs to pilot the configuration of customisable products. These tools usually render FMs in an *explorer-view* style [47, 55], as shown in the upper part of Table 1. The tick boxes in front of features are used to capture decisions, i.e., whether the features are selected or not. We now illustrate the FM abstract syntax more concretely on our working example.

2.2 *Working Example*

Audi is a German car manufacturer. Nowadays, Audi offers 12 different model lines, each available in different body styles, each broken down in different models. This paper will focus on the *Audi A3*, with the *sportback* body style. An example of its configurator in action is shown in Fig. 1. The two FMs in Fig. 2 are samples reverse engineered from the car configurator[1] for the A3 and RS3 models.

Although similar, these models show very different options to customers. The features hidden by the configurator appear in light grey. This, however, does not indicate that the value of these features is not set. It rather means that customers

[1]Reverse engineered from http://configurator.audi.co.uk/ on January 20th, 2012. Some labels were shortened for conciseness.

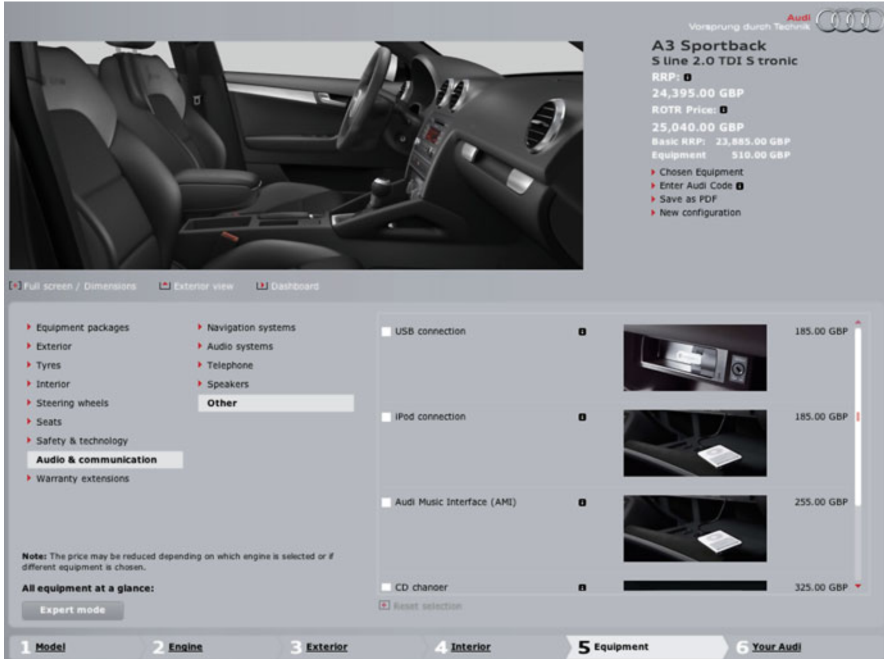


Fig. 1 Screenshot of Audi A3 configurator

cannot *manually* set their values. In Fig. 2a for instance, none of the *Engine* features are available. Yet, the RS3 has a *Quatro* drive train and a *Petrol* engine. This practice resembles the inactivation of features in operating systems configurators such as those used for Linux and eCos [13]. The isolation of visible from hidden features is thus a first possible criterion to separate concerns.

The second criterion is determined by the steps in the configuration process. As Fig. 1 shows, the configuration follows a number of steps starting from *1. Model*, going through *5. Equipment*, and ending at *6. Your Audi*. This decomposition is illustrated in Fig. 2 by the coloured areas. In contrast to the first criteria, these views are used to progressively disclose the options.

The Audi configurator, like many others (e.g. Linux and eCos [66]), rely on *ad hoc* solutions that usually do not come with a proof of completeness and correctness and can hardly be reused from one domain to the other. A general and formal foundation for separation of concerns in FBC is necessary. This paper proposes a retrospective on this SoC in FBC and discusses the complementary combination of views and slices to achieve flexible and reliable SoC. Without delving into formal developments, it provides a frame of reference to specify, verify and visualise concurrent concerns on an FM.

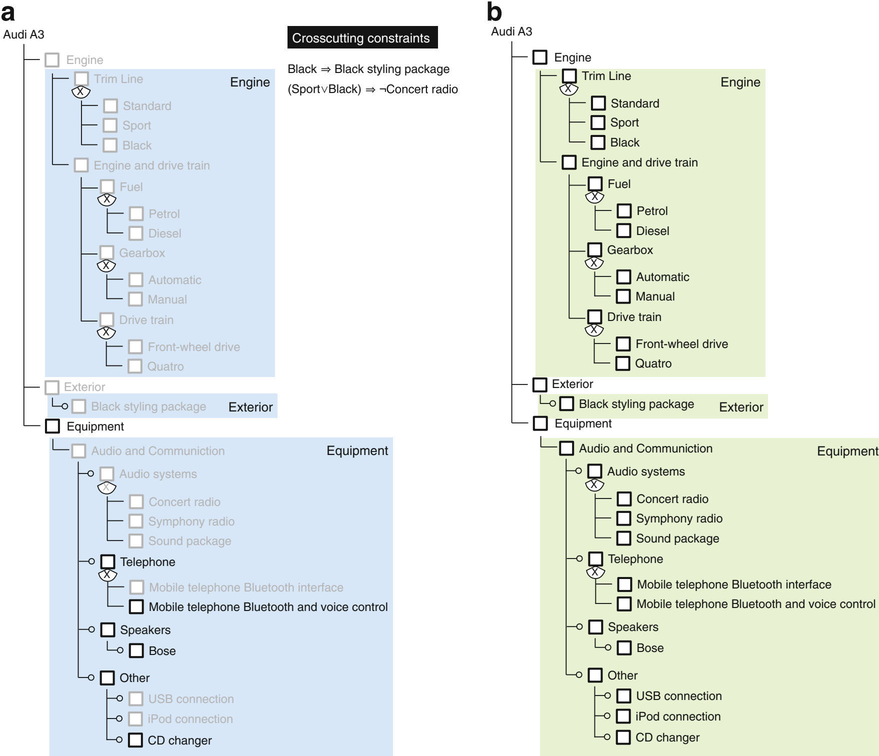


Fig. 2 Two FMs of Audi A3 model line. (**a**) Sample FM for the Audi RS 3, (**b**) Sample FM for the Audi A3

3 Concerns and Their Separation: Retrospective

A major limitation of current FM languages is that they are found not to scale well when applied to realistic SPLs. In real projects it has been reported that maintaining a single large FM for the entire system may not be feasible [26].

Firstly, FMs are increasingly larger with possibly thousands of features related by numerous complex constraints. As an extreme case, the variability model of Linux exhibits 6,000 + features [60]. Secondly, various concerns of an SPL, handled by several stakeholders (or even different organizations), should be properly modelled and managed. As a result, FMs quickly become too complex to be understood and managed by practitioners. In FBC context, it is very hard for a practitioner to consider thousands of configuration options as a whole.

The principle of SoC points to an effective way to manage the size and complexity of FMs. On the one hand, several FMs may be originally separated and combined, for instance, when engineers describe the variability of modular systems (e.g., software components or services [5]), when independent suppliers

describe the variability of their different products in software supply chains [22,33], or when a multiplicity of SPLs must be combined [26, 32]. On the other hand, it may be the intention of an SPL practitioner to modularise the variability description of the system according to different criteria or concerns such as external vs. internal variability [7,50,54], abstraction layers [41] or views tailored for a specific stakeholder [36].

The problem of SoC in FMs has been extensively reported in the literature, but there is no consensus on how best to separate and compose these concerns. In this paper, we focus on the *separation of concern* problem. This section highlights key achievements in this domain with a particular emphasis on *views*, which have been repeatedly advocated as a means to solve scalability and configuration issues.

3.1 Variability Modelling

Dealing with real-world problems implies dealing with multiple stakeholders with different and often inconsistent perspectives. Viewpoint-based approaches have been around for nearly two decades and address exactly those issues. They mainly support the identification, structuring, reconciliation and co-evolution of heterogeneous requirements [28, 51]. They have been studied mostly by the requirements engineering (RE) community. They are more concerned with the identification and reconciliation of viewpoints than with the specification and generation of viewpoint- (or concern-) specific views on an artefact like the FM in our case. Viewpoint-based RE techniques are not specific to SPLE. Still, viewpoint-based techniques can be used *upstream* of variability modelling to help build a consistent FM from heterogeneous viewpoints. More specific to variability modelling, Grünbacher et al. [31] outline the challenges that arise when heterogeneous stakeholders are involved in the modelling of large FMs.

The identification of stakeholders is also a problem studied in RE [29]. We refer the reader to [30] for a general introduction to stakeholder identification and ways to structure and trace their contributions. Directly related to feature modelling, Bidian et al. [14] identify stakeholder profiles through the tasks appearing in goal models which are subsequently linked to the features realising them.

3.2 View Specification

Early attempts to manage the complexity of FMs [40, 41] were mainly concerned with separating user-oriented from technical features. For this, simple techniques were used, namely annotation and layering of the FM, but those remained informal and were not used to generate views or for configuration. In OVM [53], a similar distinction was proposed between internal and external variability, but had the same limitations as the aforementioned approaches.

Zhao et al. [67] group features according to stakeholder profiles and other typical concerns. A major limitation is that they do not display decomposition operators in views, which greatly simplifies the problem at the expense of completeness. Features in views are physically duplicated and mapped to features of the FM. The resulting links are represented as constraints between the views and the FM.

Researchers developed SoC techniques for FMs that reflect organisational structures and tasks. Reiser et al. [56] address the problem of representing and managing FMs in SPLs that are developed by several companies, as is common, for example, in the automotive industry. They propose to use several FMs and structure them hierarchically. This way, each of them can be managed separately by one of the partner companies. Local changes are then propagated to other FMs through the hierarchy. Hierarchical decomposition in SPLs was also studied by Thompson et al. [62], although not in relation to FMs.

Clarke et al. [18] introduce a formal theory of views for FMs, where a view is defined as a disjoint set of features and *abstractions*. An abstraction encapsulates a set of features hidden behind a label meaningful to the user. They formally define compatibility properties between views and their reconciliation, i.e., combination. To preserve the genericity of their mathematical model, the authors reason exclusively in terms of features independently of the structure and constraints imposed by the FM. As a result, they do not discuss the concrete specification, rendering and configuration of views on an FM.

3.3 Configuration

Reiser et al. [56] along with Mannion et al. [44] discuss how multiple views affect the structure of the FM and configuration with a particular focus on decision propagation and conflict resolution [44]. Unlike other approaches that only consider the selection/deselection of features, they address changes to the structure of views that are propagated back to the original FM. To resolve conflicts that can happen during the merge of concurrent changes, they propose a list of conflict resolution rules within views. They thus focus on resolving conflicts among changes to the content of the FM rather than conflicts between configuration decisions.

Batory et al. [10] have worked on multi-dimensional SoC where a dimension is a set of features addressing a particular concern. They use a so-called origami matrix to describe the relationships between features across the dimensions. Their approach does not aim to generate views but rather to compose features (described separately) along each dimension.

Czarnecki et al. [23] have introduced multi-level staged configuration as a way of organizing FBC as a sequence of stages. This idea was later formalised [20] and extended [35] to deal with arbitrarily complex configuration processes (not only purely sequential ones). Mendonça et al. [46, 48] suggest configuration spaces (similar to views) as a means to support collaborative product configuration. They also provide algorithms to automatically generate a configuration plan out of an FM

and a set of configuration spaces. Although these and related [50,63] approaches are automatable and readily applicable to configuration, they remain limited to a single "tyrannical" decomposition scheme [61] (e.g., stages or workflow activities) which must be decided in advance.

Over the years, various interactive FBC environments have been developed (e.g., [8,42,45,55]). Based on formal semantics, these tools use solvers (e.g., SAT, BDD and CSP) to propagate decisions throughout the FM and ensure the global consistency of the final product. Commercial FBC tools (e.g., [42,55]) also offer integration with popular modelling environments like IBM Rational or Simulink. Traditionally, FBC tools assume that there exists a single monolithic FM and do not account for configuration processes that are distributed among various stakeholders who have specific concerns and who intervene at different moments [46, 48]. Without the appropriate support, FBC can become very cumbersome and error-prone, e.g., if a single stakeholder has to make decisions on behalf of all others [48].

4 Separating Concerns in Feature Models

4.1 Views

4.1.1 Basic Definition

Separating concerns requires the ability to specify the parts of the FM that are of interest and the person(s) who can configure it. In order to achieve this, the FM can be augmented with a set V of views, each of which consists of a set of features. Formally, a *multi-view FM* is defined as follows:

Definition 2 (Multi-view FM [36]). A multi-view FM m is a tuple $(N, r, \lambda, DE, \Phi, V)$ where $V = \{v_1, v_2, \ldots, v_n\}$ is the multiset of views such that:

- N, r, λ, DE, Φ conform to Definition 1;
- $\forall v_i \in V \bullet v_i \subseteq N \wedge r \in v_i$.

Therefore, for any concern that requires only partial knowledge of the FM, such as a profile, a view can be defined. We also consider that the root is part of each view. V is a multiset to account for duplicated sets of features.

4.1.2 View Specification

We distinguish between two ways of specifying views. With *extensional definitions*, the features that appear in a view are enumerated, or tagged so as to indicate the view to which each of the features belongs. A drawback is that the process of enumerating and tagging can be time-consuming and error-prone without appropriate tool support.

With *intensional definitions*, the features in a view are defined according to a query defined on the FM. For instance, the tree structure of the FM can be exploited by languages like XPath to specify the views. A major drawback of intensional definitions is that textual languages may not be as intuitive as graphical approaches for casual users. Furthermore, it is harder to maintain consistency between the FM and the textual expressions when the diagram evolves without proper tool support.

Having said that, extensional and intensional definitions can be used together in practice. Textual expressions corresponding to intensional specifications could be generated from a graphical view definition tool. Conversely, it is possible to generate feature tags from textual expressions and link them to the features in the expression. These links can then be used to trace changes from the FM back in the expression. This allows us to overcome the limitations of both extensional and intensional definitions. In the following discussions, we refer to features contained in views irrespective of the specification method.

4.1.3 View Coverage

An important property that should be guaranteed by an FBC system is that all configuration questions are eventually answered [20]. In a multi-view context, one may consider enforcing the following condition.

Definition 3 (Sufficient coverage condition [36]). For a view v of a multi-view FM m the sufficient coverage condition is:

$$\bigcup_{v \in V} v = N$$

Intuitively, this means that all the features appear in at least one view, hence no feature can be left undecided.[2] Although sufficient, this is not a necessary condition because some decisions can usually be deduced from others.

A *necessary condition* can be defined in terms of propositional defineability [43]. It is necessary to ensure that the decisions on the features that do not appear in any view can be inferred from (i.e., are *propositionally defined by*) the decisions made on the features that are part of the view. In the following definition, $\mathit{defines}(F, f)$ denotes the propositional definably of f by F.

Definition 4 (Necessary coverage condition [36]). For a view v of a multi-view FM m the necessary coverage condition is:

$$\forall f \notin \bigcup_{v \in V} v \bullet \quad \mathit{defines}(\bigcup_{v \in V} v \, , f)$$

[2]Note that the complete view coverage is usually assumed by multi-view approaches (e.g. [48]).

defines can be evaluated by translating the FM into an equivalent propositional formula (done in linear time [58]) and by applying the SAT-based algorithm described in [43]. Although this check is NP complete in theory, it is not expected to be a problem in practice, since SAT solvers can handle FMs with thousands of features.

Features in $N \setminus \bigcup_{v \in V} v$ that do not satisfy the above condition will have to be integrated in existing views, or extra constraints will have to be added to determine their value.

In application domains such as operating systems, features such as those used for calculating the boot entry to use are hidden from users [13] and may not be visible in any view. In such cases, the verification of the necessary condition determines whether the value of the hidden features can be derived from the features in the views.

However, in cases such as the Audi configurator (Sect. 2.2), these two conditions are too strict. Assuming that a view only contains the features relevant to a customer, it will naturally not contain the hidden features. In this particular case, some hidden features might not be decided upon. Some existing configurators, such as the one of Linux, nullify these features. In this context, the necessary coverage condition has to be adapted such that one only checks features that are neither in $\bigcup_{v \in V} v$ nor in the nullified features.

4.2 Visualisation

Although views are abstract, they have to be made concrete to be used during FBC. A concrete view is called a *visualisation*. A visualisation strives to find a compromise between not showing in a view features that do not belong to the view, and showing features that do not belong to the view but indirectly provide context for features that should be shown. In Fig. 2, for instance, feature Y is in the view (darker area), but its parent feature A is not.

To tackle this problem of view rendering, we have observed the practice of developers (see [37] for more details about the case study and our experience with PloneMeeting) and discussed with them alternative visualisations. Our discussions included the relative merits of the approaches suggested in [48, 67], and the filtering mechanisms provided by tools such as pure::variants [55], and kernel configurators for operating systems (e.g., `xconfig` for Linux and `configtool` for eCos [27]). These tools provide simple filtering or search mechanisms that are similar to views on an FM. In these cases, a filter is a regular expression on the FM. Any feature matching the regular expression is displayed typically without any control on the location of the feature in the hierarchy. Interestingly, all these approaches produce purely graphical modifications (e.g., by greying out irrelevant features) whereas cardinalities are not recomputed.

The main outcome of our investigation is a set of four complementary visualisations offering different levels of details, as depicted in Fig. 3. The darker area

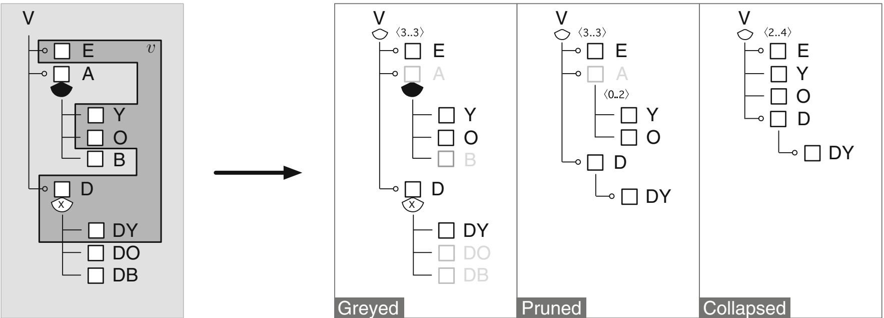


Fig. 3 Three alternative visualisations of FM views: greyed, pruned and slice/collapsed [36]

defines a specific view of the FM, called v. These views were built to present information on a *need-to-know* basis. The amount of information displayed can be regulated, while providing enhanced control over access rights. For instance, there is always a standardised configuration menu that can display the position of the feature in the hierarchy and hide unavailable options. On the other hand, in a critical application, features outside a view may have to be protected as trade secrets. Therefore, visualisations not only can provide convenient representations of a view, but they can also restrict the information a stakeholder can access.

Figure 3 illustrates the alternative visualisations of FM views we propose:

- The *greyed* visualisation is a mere copy of the whole FM except that the features that do not belong to the view are greyed out (e.g., A, B, DO and DB). Greyed features are only displayed but cannot be manually selected/deselected.
- In the *pruned* visualisation, features that are not in the view are pruned (e.g., B, DO and DB) unless they appear on a path between a feature in the view and the root, in which case they are greyed out (e.g., A).
- In the *collapsed* visualisation, all the features that do not belong to the view are pruned. A feature in the view whose parent or ancestors are pruned is connected to the closest ancestor that is still in the view. If no ancestor is in the view, the feature is directly connected to the root (e.g., Y and O).
- The *slice* visualisation is similar to the *collapsed* visualisation except that it takes cross-tree constraints into consideration. Consequently, decomposition operators might be altered to preserve the correctness of these constraints.

Generating visualisations, from an FM and a view, is a form of FM transformation.

Definition 5 (View visualisation). The visualisation of a view v is the transformation of the original FM into a new FM $d_v^t = (N_v^t, r, \lambda_v^t, DE_v^t, \Phi)$, where t, the type of visualisation, can take one of four values: g (greyed), p (pruned), c (collapsed) and s (slice).

The greyed visualisation is the simplest case because there is no transformation beyond the greying of each feature $f \notin v$ (i.e., $d_v^g = d$). The transformations

for the pruned and collapsed visualisations, on the other hand, filter nodes, remove dangling decomposition edges and adapt the cardinalities accordingly.

4.2.1 Pruned Visualisation

N_v^p, the set of features in this visualisation, is the subset of N limited to features that are in v or have a descendant in v. The definition uses DE^+, the transitive closure of DE. Based on N_v^p, we remove all dangling edges, i.e., those not in $N_v^p \times N_v^p$ to create DE_v^p.

Transformation 1 (Pruned visualisation [36]).
The transformations applied to the FM to generate the pruned visualisation are:

$$\begin{aligned} N_v^p &= \{n \in N | n \in v \vee \exists f \in v \bullet (n, f) \in DE^+\} \\ DE_v^p &= \{DE \cap (N_v^p \times N_v^p)\} \\ \lambda_v^p(f) &= (mincard_v^p(f), maxcard_v^p(f)) \end{aligned}$$

In order to compute the new cardinalities $\lambda_v^p(f)$, $mincard_v^p(f)$ and $maxcard_v^p(f)$ are defined as follows:

$$\begin{aligned} mincard_v^p(f) &= max(0, \lambda(f).min - |orphans_v^p(f)|) \\ maxcard_v^p(f) &= min(\lambda(f).max, |children(f)| - |orphans_v^p(f)|) \end{aligned}$$

where $orphans_v^p(f) = children(f) \setminus N_v^p$ i.e., the set of children of f that are not in N_v^p. $\lambda(f).min$ and $\lambda(f).max$ represent the minimum and maximum values of the original cardinality, respectively. For the minimum, the difference between the cardinality and the number of orphans can be negative in some cases, hence the necessity to take the maximum between this value and 0. The maximum value is the maximum cardinality of f in d if the number of children in v is greater. If not, the maximum cardinality is set to the number of children that are in v.

4.2.2 Collapsed Visualisation

The set of features N_v^c of this visualisation is simply the set of features in v. The consequence on DE_v^c is that some features have to be connected to their closest ancestor if their parent is not part of the view.

Transformation 2 (Collapsed visualisation [36]). *The transformations applied to the FM to generate the collapsed visualisation are:*

$$\begin{aligned} N_v^c &= v \\ DE_v^c &= \{(f, g) | f, g \in v \wedge (f, g) \in DE^+ \wedge \nexists f' \in v \bullet ((f, f') \in DE^+ \wedge (f', g) \in DE^+)\} \\ \lambda_v^c(f) &= (mincard_v^c(f), maxcard_v^c(f)) \end{aligned}$$

The computation of cardinalities $\lambda_v^c(f)$ is slightly more complicated than in the pruned case. Formally, $mincard_v^c(f)$ and $maxcard_v^c(f)$ are defined as follows:

$$\begin{aligned} mincard_v^c(f) &= \sum min_{\lambda(f).min}(ms_min_v^c(f)) \\ maxcard_v^c(f) &= \sum max_{\lambda(f).max}(ms_max_v^c(f)) \end{aligned}$$

where

$$\begin{aligned} ms_min_v^c(f) &= \{mincard_v^c(g) | g \in orphans_v^c(f)\} \uplus \{1 | g \in children(f) \setminus orphans_v^c(f)\} \\ ms_max_v^c(f) &= \{maxcard_v^c(g) | g \in orphans_v^c(f)\} \uplus \{1 | g \in children(f) \setminus orphans_v^c(f)\} \end{aligned}$$

The multisets $ms_min_v^c(f)$ and $ms_max_v^c(f)$ collect the cardinalities of the descendants of f. The left part of the union[3] recursively collects the cardinalities of the collapsed descendants whereas the right side adds 1 for each child that is in the view. The $\lambda(f).min$ minimum values of the multiset are then summed to obtain the minimum cardinality of f. The maximum value is computed similarly.

4.2.3 Slice Visualisation

We revisit here a technique called *slicing* that, given an FM (typically large), produces a new, smaller FM containing only a subset of features of the input FM. We show that the slicing can be used to synthesize visualisations.

The overall idea behind FM slicing is similar to program slicing [65]. Program slicing has been successfully applied in computer programming and aims at simplifying or abstracting programs by focusing on selected aspects of semantics. Program slicing techniques usually proceed in two steps: the subset of elements of interest (e.g., a set of variables of interest and a program location), called the slicing *criterion*, is first identified; then, a *slice* (e.g., a subset of the source code) is computed. In the context of FMs, we define the slicing criterion as a set of features considered to be pertinent by an SPL practitioner while the slice is a new FM (see Transformation 3).

Slicing Semantics. The major preoccupation for an SPL practitioner is the legal combination of features (configurations) defined by an FM. The same observation applies when decomposing the FM into smaller concerns. We want to guarantee semantic properties of smaller parts, i.e., in terms of set of configurations. Nevertheless, several FMs, yet with different hierarchies, can represent a given set of configurations. Therefore, the semantics of the slicing operator is defined both in terms of set of configurations and feature hierarchy (see Transformation 3).

Transformation 3 (Slice visualisation [4]). *We define slicing as an operation on FM, denoted* $\Pi_{N_v^s}(d) = d_v^s$ *where* N_v^s *is a set of features, called the slicing criterion, and* d_v^s *is a new FM, called the slice.*

[3] $\uplus$ is the union on multisets.

The result of the slicing operation is a new FM, d_v^s*, such that:*

- *Feature hierarchy: Features of the hierarchy include the slicing criterion of the original FM while features are connected to their closest ancestor if their parent feature is not part of the slice FM. It corresponds to the feature hierarchy defined for the collapsed visualisation (see Transformation 2).*
- *Configuration semantics: The valid configurations,* $[\![d_v^s]\!]$*, one could infer from a slice are actually the valid configurations of the original FM, when looking only at the slicing criterion features* N_v^s*. Formally, the projected*[4] *set of configurations is defined as* $[\![d_v^s]\!] = [\![d]\!]_{|N_v^s}$*.*

It should be noted that the hierarchy of the slice FM corresponds to the hierarchy defined for the collapsed visualisation (see the right hand side of Fig. 3). In the following, we will describe an algorithm to synthetize automatically such visualisations.

Automated Slice Synthesis. Our previous experience has shown that *syntactic* strategies have severe limitations to accurately represent a given set of configurations (as expected by Transformation 3), especially in the presence of cross-tree constraints [2]. The same observation applies for the slicing operation so that we reason directly at the *semantic* level. The key idea of the proposed algorithm is to *(i)* compute the propositional formula representing the projected set of configurations, and then to *(ii)* apply satisfiability techniques to construct a complete FM (including variability information and cross-tree constraints) using the formula. A major difference with previous works [24, 60] that propose to synthetize FMs from propositional formulae is that the feature hierarchy of the resulting FM can be determined and computed (see Transformation 3).

Formula Computation. Let $d_v^s = \Pi_{N_v^s}(d)$. The propositional formula ϕ_s corresponding to d_v^s can be defined as follows:

$$\phi_s \equiv \exists\, f_1, f_2, \dots f_{m'}\ \phi$$

where $f_1, f_2, \dots f_{m'} \in (N \setminus N_v^s) = N_{removed}$ and ϕ is the encoding of d as a propositional formula. The propositional formula ϕ_s is obtained from ϕ by *existentially quantifying* out variables in $N_{removed}$. Intuitively, all occurrences of features that are not present in any configuration of d_v^s are removed by existential quantification[5] in ϕ.

From Formula to FM. From the propositional formula ϕ_s, several FMs can be synthesised [60]. In our case, though, we already *know* what the resulting hierarchy

[4]For two given sets A and B, we note $A_{|B}$ the projection of A on B such that: $A_{|B} \triangleq \{a' \mid a \in A \wedge a' = a \cap B\} = \{a \cap B \mid a \in A\}$

[5]Existential quantification is defined as the substitution of a Boolean variable *ft* to True and False values. Formally: $\exists ft\ \phi =_{def} \phi_{|ft} \vee \phi_{|\bar{ft}}$ where $\phi_{|ft}$ (resp. $\phi_{|\bar{ft}}$) denotes the assignment of *ft* to True (resp. False) value in ϕ.

is. Our algorithm exploits this information. We first compute the hierarchy, we then set the variability information (mandatory/optional, Xor and Or-groups) and finally the constraints (bi)-implies/excludes/others.

Mandatory and Feature Groups. At this step, all features, except root, are considered optional. We compute the binary implication graph, noted BIG_s, of the formula ϕ_s over N_v^s.

BIG_s is a directed graph $G = (V, E)$ formally defined as:

$$V = N_s \qquad E = \{(f_i, f_j) \mid \phi_s \land f_i \Rightarrow f_j\}$$

We use BIG_s to identify biimplications and thus set mandatory features together with their parents. For feature groups, we reuse the prime implications method proposed in [24], so that we can identify Or- and Xor-groups. An important issue is that a feature may be candidate to several feature groups (which is not allowed by FMs). Therefore some feature groups are dismissed so that FMs are well formed. We use the original FM to retrieve initial feature groups (see details in [1]).

Constraints. The set of implies constraints can be deduced by removing edges of BIG_s that are already expressed (e.g., parent–child relations). For the purpose of conciseness, some implies constraints can be transformed into equivalence relations (e.g., $A \Rightarrow B \land B \Rightarrow A$ can be transformed into $A \Leftrightarrow B$). Similarly, excludes constraints are produced by computing the binary exclusion graph of ϕ_s over N_v^s. Excludes constraints that were not chosen to be represented as an Xor-group are added. When adding constraints, we control that the constraint is not already induced by the FM. At this end, it should be noted that the FM may still be an over approximation of ϕ_s.[6] Using standard propositional logics techniques, we can calculate the complement between the current set of configurations represented by the FM and the expected set of configurations of the slice FM. The complement can be recovered, for instance, as a conjunction of propositional constraints.

4.2.4 Properties and Comparison

We now discuss properties of the slicing technique and the transformations described above regarding their ability to produce visualisations.

Semantic Preservation. It is important to demonstrate that the visualisations preserve a form of semantic equivalence with the original FM. We define the semantic equivalence in terms of the set of configurations characterised by the original FM and the projected set of configuration characterised by the collapsed visualisation.

[6] In [24], the authors characterised the limited expressiveness of FMs compared to propositional logic.

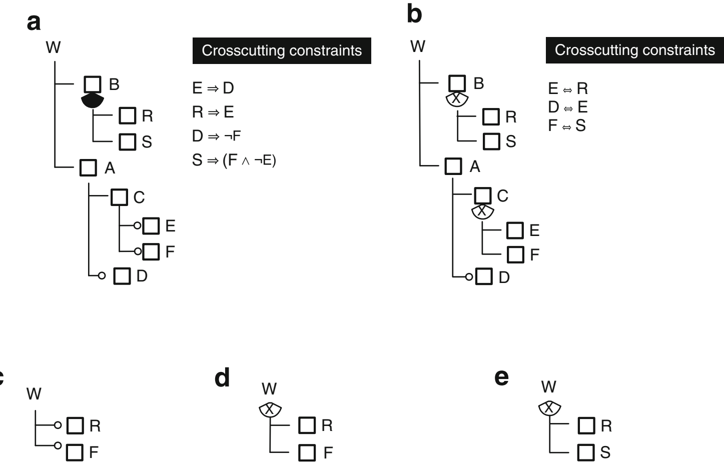


Fig. 4 Collapsed visualisations (transformation and slicing). (**a**) Original FM, (**b**) Corrected FM, (**c**) Collapsed view (transformation), (**d**) Collapsed view (slicing), (**e**) Corrective capabilities (slicing)

Accuracy of Visualisations. As demonstrated in [36] for FMs without constraints, the greyed and pruned visualisations preserve the semantic equivalence. Using the syntactical transformations though, the semantic equivalence in the collapsed visualisation does not hold. Take the simple counter-example shown in Fig. 4a and the collapsed visualisation of view v depicted in Fig. 4c. A valid configuration of the collapsed visualisation would be $\{W, R, S\}$. However, that configuration is not valid in the FM since R and S must not appear together in a configuration. This shows that the transformation that produces the collapsed visualisation does not preserve the semantics of the FM: The collapsed visualisation is an under-constrained FM. This is, however, not a limitation in practice. When FBC is assisted by a solver, the solver preserves the global consistency of the FM. It thereby prevents possible errors induced by the under-constrained model presented in the collapsed visualisation. In the counter-example in Fig. 4c for instance, the selection of R in the view will automatically entail the deselection of S, even though the recomputed cardinality does not enforce that propagation.

Using the slicing technique, the collapsed visualisation respects by construction of the semantic equivalence. It exactly corresponds to Transformation 3 that specifies the relationship between the original FM and the slice FM. For example, the slicing is able to enforce that R and F features are mutually exclusive (see Fig. 4d).

Assumptions about Input FMs. In [36], the correctness of transformations for the three kinds of visualisations has been shown for FMs without cross-tree constraints. The reason is that arbitrary cross constraints can have an influence on the visualisations. By reasoning directly at the semantic level, the slicing technique

is applicable to any kind of FMs, including arbitrary propositional constraints. However, the slicing technique does not support generic $< i..j >$ cardinality.

Corrective Capabilities. Due to the presence of constraints, an input FM may contain *anomalies*, for example, dead features, false optional features, or redundancies (see [12]). The slicing algorithm ensures, by construction, that there is no dead feature, correctly detect mandatory features and avoids redundancy of constraints. Therefore the slicing operator can be used as an *automated technique to correct* anomalies of FMs while preserving the original set of configurations and feature hierarchy. Two examples are given in Fig. 4b, e. Moreover the corrective modifications applied to the original FM can be detected and reported to SPL practitioner so that they can understand the anomalies.

Impact on Other Kinds of Visualisation. Another application of the corrective property is that the slicing technique can be used to produce more accurate greyed and pruned visualisations (e.g., by correcting wrong cardinalities). For the greyed visualisation, the slicing is first applied on the original FM, using the whole set of features as slicing criterion (see Fig. 4e for a corrected FM) while some features are then greyed out. For the pruned visualisation, the slicing is first applied on the original FM, using the set of features N_v^p of Definition 1 as slicing criterion. Features are then greyed out in line with the definition of pruned visualisation.

Performance. Synthesising views at the semantic level, though more powerful, has a cost. Satisfiability techniques that reason over propositional formula are used and can be realized using either satisfiability (SAT) solvers or binary decision diagrams (BDDs) [24, 60].

BDD-Based Implementation. A BDD can be seen as a compact representation of a propositional formula. BDDs are known to efficiently compute the existential quantification of a propositional formula in at most polynomial time with respect to the sizes of the BDDs involved. Moreover, as shown in [24], BDDs can be used to synthesize an FM in polynomial time regarding the size of the BDD representing the input propositional formula. The primary limitation of a BDD-based implementation is related to the space complexity: *(i)* as shown in [21], computing the BDD of an FM containing more than 2,000 features is intractable; *(ii)* from our experiments, the synthesis of FMs has practical limits (up to 800[7] features) mainly due to the cost of computing Or-groups.

SAT-Based Implementation. BDDs do not scale for very large FMs (e.g. Linux FM that has more than 6,000 features). In [60], She et al. proposed to rely on SAT solvers (rather than BDDs as in [24]) and reported that the use of SAT solvers is significantly more scalable. As SAT solvers require the formula to be in conjunctive normal form (CNF). To avoid the exponential explosion of disjunctive clauses, we developed

[7] Janota et al. reported that the BDD-based algorithm proposed in [24] scales up only for FMs with 300/400 features [39] but did not use the heuristics proposed in [21] that reduce the size of BDDs.

specific techniques and some heuristics to determine the order in which existential quantification should be applied [4]. Using the slicing technique on generated and real-world FMs, we found that: *(i)* computing the propositional formula is almost instantaneous for all FMs of SPLOT (less than one second, whatever the size of the slicing criterion is); *(ii)* the SAT-based implementation scales for a number of features (#*features*) lesser than 10,000 whatever the size of the slicing criterion is, but not for the Linux FM; *(iii)* the order in which the features are existentially quantified is of prior importance: We observe scalability issues when quantifying first the features that are at the top of the feature hierarchy for #*features* $\geq$ 2,000; *(iv)* for very large FMs (#*features* $\geq$ 5,000), the computation time is inadequate for an interactive use of the slice operator (up to 20 min).

Summary and Comparison. On the one hand, the slicing technique is more general (i.e., applicable to any kind of FMs and propositional constraints) and accurate than the syntactical transformations. In particular the collapsed visualisation is no longer an under-approximation of the projected set of configurations (see example in Fig. 4). On the other hand, some limitations remain: Lack of support for generalised cardinality and performance issues. As a result, a *tradeoff* should be found when producing collapsed visualisations. In this case, the slicing or the syntactical transformations can be chosen on demand, i.e., regarding the kind of visualisations, the number of features, the presence of cross-tree constraints, etc. For other kinds of visualisations (greyed, pruned), anomalies can be first corrected, the syntactical transformations being applied afterwards. The consistency of the FM for under-constrained collapsed views can be maintained by reasoning about the complete FM in the back-end.

5 Tool Support

Armed with these definitions, we now present two complementary tools that support view management. The first tool has been developed in the context of FBC while the second tool targets the large-scale management of FMs through a dedicated language. They both support view *specification* with two similar solutions (i.e., XPath and a specific textual notation) and can be connected together.

5.1 View Creation and Visualisation in SPLOT

The tool support developed for multiview FBC builds upon SPLOT [49]. To provide efficient interactive configuration, SPLOT relies on a SAT solver (SAT4J) and a BDD solver (JavaBDD). Their reasoning abilities enable error detection and decision propagation. SPLOT was chosen because it offers robust support for FBC, it is easy to extend, and the existing repository of FMs is an excellent testbed

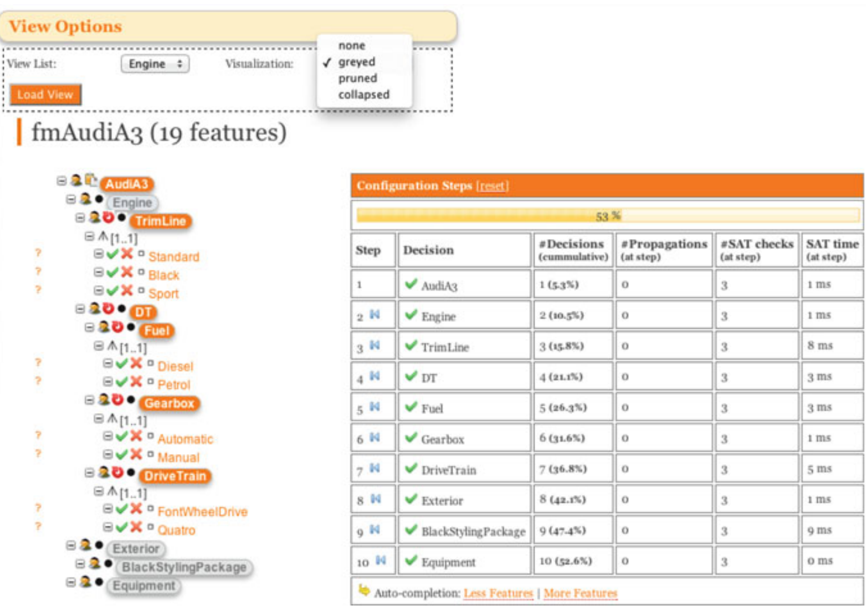


Fig. 5 Configuration view of the *Engine* with the greyed visualisation in SPLOT

for multiview FMs. All our extensions to SPLOT are available online.[8] The three extensions supporting multiview FBC are briefly introduced below.

The first extension enables view creation with XPath expressions. An online evaluator checks that the XPath expression is correct and shows the results of its evaluation. Moreover, the completeness of the views can be checked interactively and the features that are not covered, if any, are returned.

The actual configuration of a view is provided by the second extension. The extension allows stakeholders to select (1) the view to configure and (2) the visualisation. In Fig. 5, the view of the *Engine* is selected and the pruned visualisation is activated. Note the greyed *Exterior*, *Equipment* and *BlackStylingPackage* features that can neither be selected nor deselected. The stakeholder can switch freely from one visualisation to another as he/she configures his/her view without losing the decisions that were already made. This way, we *dynamically combine* the advantages of the three visualisations and leave complete freedom to the stakeholder to choose the one(s) that best fit(s) her preferences. The table on the right monitors the status of the current configuration. Basically, it tells what features have been selected or deselected, and which decisions were propagated. As explained in Sect. 4.2, the solver reasons about the full FM and not only about an individual view. Thereby, the decision to select or deselect a feature in the view is propagated in the complete model—keeping the global configuration consistent.

[8] http://www.splot-research.org/extensions/fundp/fundp.html.

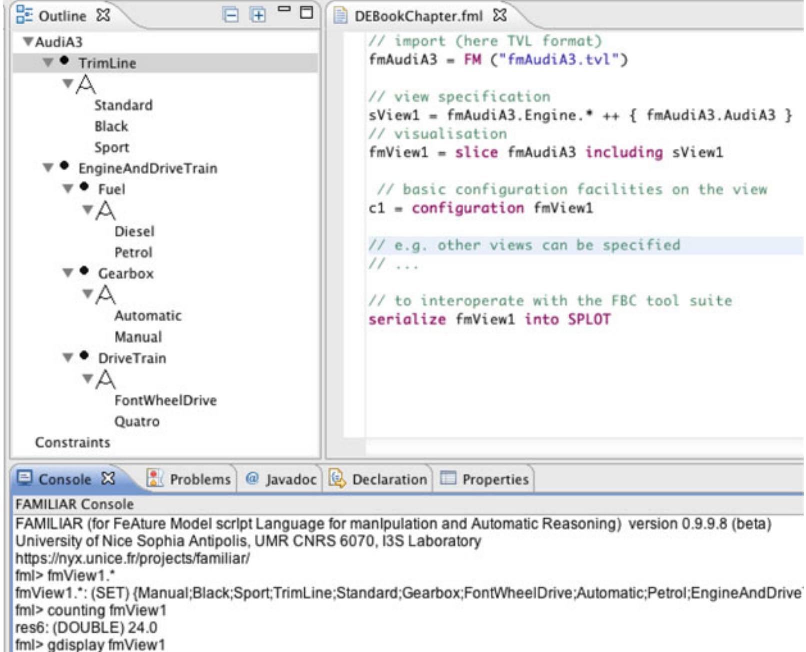


Fig. 6 Slicing example and interoperability in the FAMILIAR environment

The third extension provides basic support for multi-user concurrent configuration. At the time being, it only enables synchronous configuration. To prevent conflicting decisions, a configuration session manager is used. Its role is (1) to maintain a mutual exclusion on the configuration engine so that only one user can commit a decision at a time and (2) to notify all users about a decision and about the results of the propagation.

5.2 *Slicing and FAMILIAR*

As seen in the previous sections, the slicing operator can be used to produce collapsed visualisations. The operator is part of FAMILIAR (for *FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) a domain-specific language for large-scale management of FMs [3].

Examples of the syntax of the slicing operator are given in Fig. 6. The set of features that constitutes the slicing criterion can be specified either by inclusion (keyword: including) or exclusion (keyword: excluding). Intentional definitions of views can be specified in the style of XPath. Off-the-shelf SAT solvers (i.e., SAT4J) and BDD library (i.e., JavaBDD) are internally used. The slicing operator produces a new FM that can be manipulated using variables. FAMILIAR also includes functions for composing FMs, editing FMs (e.g., renaming and removal of features),

reasoning about FMs (e.g., validity, comparison of FMs) and their configurations (e.g., counting or enumerating the configurations in an FM). FAMILIAR comes with an Eclipse-based environment that is composed of textual editors, an interpreter that executes FAMILIAR scripts, and an interactive front-end. The FMs can be serialised in different formats (SPLOT, FeatureIDE, a subset of TVL, etc.).

Thanks to the integration with SPLOT, we can realise scenarios in which an FM is first corrected using the slicing operator of FAMILIAR and then used in the FBC web environment.

6 Conclusion

Feature models (FMs) are widely used to represent the valid combination of features supported by a family of systems in a given domain. In real variability-intensive systems, many *concerns* have to be considered. These concerns are related in a variety of ways and there can be thousands of features whose legal combinations are governed by many and often complex rules. It has been observed that configuring or maintaining a single large FM may not be feasible [26, 54]. Views (as a simplified representation of an FM tailored for a specific stakeholder, role or task) have been repeatedly identified as a possible solution to the scalability and configuration issues of FMs.

In this chapter, we reviewed concerns and their separation in FMs, revisiting the concept of view, and discussing the major results in the literature. Then, we delved into the three specific problems of multi-view FMs: The *specification* of a view, the *coverage* of a set of views, and the *visualisation* of a view. Finally, we presented two tools that provide support for these three problems.

Several avenues for future work can be envisaged. The three alternative visualisations were developed to provide more flexibility to the configuration environment and more precise contextual information to the user. That improvement is, however, limited to tree-like representations of FMs. Recent advances deviate from the traditional explorer-like representations [15, 17], while others recommend dedicated configuration interfaces [16, 52]. Understanding the most suitable interfaces for multi-view will require qualitative user studies. More generally, we plan to study further the practical usage and applicability of the proposed techniques in various domains (e.g., operating systems [13, 66] and video surveillance [6]).

References

1. Acher, M.: Managing multiple feature models: foundations, language and applications. PhD thesis, University of Nice Sophia Antipolis, Nice (2011)
2. Acher, M., Collet, P., Lahire, P., France, R.: Comparing approaches to implement feature model composition. In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'10), vol. 6138 of LNCS, pp. 3–19, 2010

3. Acher, M., Collet, P., Lahire, P., France, R.: A domain-specific language for managing feature models. In: Proceedings of the Symposium on Applied Computing (SAC'11), pp. 1333–1340. ACM, New York (2011)
4. Acher, M., Collet, P., Lahire, P., France, R.: Separation of concerns in feature modeling: support and applications. In: Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD'12), pp. 1–12. ACM, New York (2012)
5. Acher, M., Collet, P., Lahire, P., Gaignard, A., Robert, F., Montagnat, J.: Composing multiple variability artifacts to assemble coherent workflows. Software Q. J. (Special issue on Quality for SPLs) **20**(3–4), 689–734 (2012). http://dx.doi.org/10.1007/s11219-011-9170-7
6. Acher, M., Collet, P., Lahire, P., Moisan, S., Rigault, J-P.: Modeling variability from requirements to runtime. In: 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2011), Las Vegas, 27–29 April 2011
7. Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., Merle, P.: Feature model differences. In: Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE'12), LNCS. Springer, New York (2012)
8. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: feature modeling plug-in for Eclipse. In: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology EXchange, pp. 67–72. ACM, Vancouver, BC (2004)
9. Apel, S., Kästner, C.: An overview of feature-oriented software development. J. Object Tech. **8**(5), 49–84 (2009)
10. Batory, D., Liu, J., Sarvela, J.N.: Refinements and multi-dimensional separation of concerns. In: Proceedings of the 9th European Software Engineering Conference (ESEC'03), pp. 48–57. ACM, New York (2003)
11. Batory, D., Liu, J., Sarvela, J.N.: Refinements and multi-dimensional separation of concerns. SIGSOFT Software Eng. Note **28**, 48–57 (2003)
12. Benavides, D., Segura, S., Ruiz-Cortes, A.: Automated analysis of feature models 20 years later: A literature reviews. Inform. Syst., Elsevier Science Ltd., **35**(6), Oxford, UK (2010). http://dx.doi.org/10.1016/j.is.2010.01.001
13. Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K.: Variability modeling in the real: a perspective from the operating systems domain. In: Proceedings of the 25th International Conference on Automated Software Engineering (ASE'10), pp. 73–82. ACM, New York (2010)
14. Bidian, C.: From stakeholder goals to product features: towards a role-based variability framework with decision boundary. In: Proceedings of the 4th International Conference on Privacy, Security and Trust (PST '06), pp. 1–5. ACM, New York (2006)
15. Botterweck, G., Thiel, S., Nestor, D., bin Abid, S., Cawley, C.: Visual tool support for configuring and understanding software product lines. In: Proceedings of the 12th International Software Product Line Conference (SPLC '08), pp. 77–86. IEEE Computer Society, Washington, DC (2008)
16. Boucher, Q., Perrouin, G., Heymans, P.: Deriving configuration interfaces from feature models: a vision paper. In: Sixth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12), pp. 37–44, 2012
17. Cawley, C., Healy, P., Botterweck, G., Thiel, S.: Research tool to support feature configuration in software product lines. In: Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), pp. 179–182. University of Duisburg-Essen, January 2010
18. Clarke, D., Proenca, J.: Towards a theory of views for feature models. In: Proceedings of the 1st International Workshop on Formal Methods in Software Product Line Engineering (FMSPLE'10), 2010
19. Classen, A., Heymans, P., Schobbens, P.-Y.: What's in a feature: a requirements engineering perspective. In: FASE'08, held as Part of ETAPS'08, pp. 16–30, 2008
20. Classen, A., Hubaux, A., Heymans, P.: A formal semantics for multi-level staged configuration. In: Proceedings of the 3rd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09), pp. 51–60, 2009

21. Czarnecki, K., She, S., Wasowski, A.: Sample spaces and feature models: There and back again. In: Proceedings of the 12th International Software Product Line Conference (SPLC'08), pp. 22–31. IEEE Computer Society, Washington, DC (2008)
22. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process Improv. Pract. **10**(1), 7–29 (2005)
23. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration through specialization and multi-level configuration of feature models. Software Process Improv. Pract. **10**(2), 143–169 (2005)
24. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: Proceedings of the 13th International Software Product Lines Conference (SPLC'07), pp. 23–34. IEEE Computer Society, Washington, DC (2007)
25. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. J. Syst. Software **74**(2), 173–194 (2005)
26. Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T.: Structuring the modeling space and supporting evolution in software product line engineering. J. Syst. Software **83**(7), 1108–1122 (2010)
27. eCos.: User Guide. http://ecos.sourceware.org/docs-latest/user-guide/ecos-user-guide.html, March 2011
28. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A framework for integrating multiple perspectives in system development. Int. J. Software Eng. Knowl. Eng. **2**, 31–58 (1992)
29. Glinz, M., Wieringa, R.J.: Guest editors' introduction: stakeholders in requirements engineering. IEEE Software **24**, 18–20 (2007)
30. Gotel, O., Finkelstein, A.: Contribution structures. In: Proceedings of the 2nd International Conference on Requirements Engineering (RE'95), pp. 100–107. IEEE Computer Society, Washington, DC (1995)
31. Grünbacher, P., Rabiser, R., Dhungana, D., Lehofer, M.: Structuring the product line modeling space: strategies and examples. In: Proceedings of the 3rd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09), pp. 77–82, 2009
32. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: Proceedings of the 12th International Software Product Lines Conference (SPLC'08), pp. 12–21. IEEE Computer Society, Washington, DC (2008)
33. Hartmann, H., Trew, T., Matsinger, A.: Supplier independent feature modelling. In: Proceedings of the 13th International Software Product Lines Conference (SPLC'09), pp. 191–200. IEEE Computer Society, Washington, DC (2009)
34. Hubaux, A.: Feature-based configuration: collaborative, dependable, and controlled. PhD thesis, University of Namur (2012)
35. Hubaux, A., Classen, A., Heymans, P.: Formal modelling of feature configuration workflow. In: Proceedings of the 13th International Software Product Lines Conference (SPLC'09), pp. 221–230. Carnegie Mellon University, San Francisco, CA, (2009)
36. Hubaux, A., Heymans, P., Schobbens, P.-Y., Deridder, D., Abbasi, E.: Supporting multiple perspectives in feature-based configuration. Software & Syst. Model. (Springer) 1–23 (2011). http://link.springer.com/article/10.1007%2Fs10270-011-0220-1
37. Hubaux, A., Heymans, P., Schobbens, P-Y., Deridder, D.: Towards multi-view feature-based configuration. In: 16th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'10), vol. 6182 of LNCS, pp. 106–112, 2010
38. Janota, M.: SAT solving in interactive configuration. PhD thesis, University College Dublin (2010)
39. Janota, M., Kuzina, V., Wasowski, A.: Model construction with external constraints: An interactive journey from semantics to syntax. In: 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'08), vol. 5301 of LNCS, pp. 431–445, 2008

40. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990
41. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: Form: A feature-oriented reuse method with domain-specific reference architectures. Ann. Software Eng. **5**, 143–168 (1998)
42. Krueger, C.: BigLever Software, Inc. http://www.biglever.com/index.html, May 2010
43. Lang, J., Marquis, P.: On propositional definability. Artif. Intell. **172**(8–9), 991–1017 (2008)
44. Mannion, M., Savolainen, J., Asikainen, T.: Viewpoint-oriented variability modeling. In: Proceedings of the 33rd International Computer Software and Applications Conference (COMPSAC'09), pp. 67–72. IEEE Computer Society, Washington, DC (2009)
45. Mendonça, M.: SPLOT. http://www.splot-research.org/, May 2010
46. Mendonça, M., Tonelli Bartolomei, T., Cowan, D.: Decision-making coordination in collaborative product configuration. In: Proceedings of the 23rd Symposium on Applied computing (SAC'08), pp. 108–113. ACM, Fortaleza, Ceara (2008)
47. Mendonça, M.: Efficient reasoning techniques for large scale feature models. PhD thesis, University of Waterloo (2009)
48. Mendonça, M., Cowan, D.D., Malyk, W., de Oliveira, T.C.: Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. J. Software **3**(2), 69–82 (2008)
49. Mendonnça, M., Branco, M., Cowan, D.: S.P.L.O.T.: software product lines online tools. In: Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA'09), pp. 761–762. ACM, New York (2009)
50. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.-Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: Proceedings of 15th International Conference on Requirements Engineering (RE'07), pp. 243–253. IEEE Computer Society, Washington, DC (2007)
51. Nuseibeh, B., Kramer, J., Finkelstein, A.: Viewpoints: meaningful relationships are difficult! In: Proceedings of the 25th International Conference on Software Engineering (ICSE'03), pp. 676–681. IEEE Computer Society, Washington, DC (2003)
52. Pleuss, A., Botterweck, G., Dhungana, D.: Integrating automated product derivation and individual user interface design. In: Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10), pp. 69–76, 2010
53. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)
54. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)
55. pure-systems GmbH: Variant management with pure::variants. http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf, 2006
56. Reiser, M.-O., Weber, M.: Managing highly complex product families with multi-level feature trees. In: Proceedings of the 14th International Conference on Requirements Engineering (RE'06), pp. 146–155. IEEE Computer Society, Washington, DC (2006)
57. Reiser, M.-O., Weber, M.: Multi-level feature trees: A pragmatic approach to managing highly complex product families. Requir. Eng. **12**(2), 57–75 (2007)
58. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Network **51**(2), 456–479 (2007)
59. Schobbens, P-Y., Heymans, P., Trigaux, J-C., Bontemps, Y.: Feature diagrams: a survey and a formal semantics. In: Proceedings of the 14th International Requirements Engineering Conference (RE'06), pp. 139–148. IEEE Computer Society, Washington, DC (2006)
60. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: Proceedings of the 33th International Conference on Software Engineering (ICSE'11), pp. 461–470. ACM, New York (2011)
61. Tarr, P., Ossher, H., Harrison, W., S; Sutton, M. Jr.: N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99), pp. 107–119. IEEE Computer Society, Washington, DC (1999)

62. Thompson, J.M., Heimdahl, M.P.E.: Structuring product family requirements for n-dimensional and hierarchical product lines. Requirements Eng. J. **8**(1), 42–54 (2003)
63. Tun, T.T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P.: Relating requirements and feature configurations: A systematic approach. In: Proceedings of the 13th International Software Product Lines Conference (SPLC'09), pp. 201–210. IEEE Computer Society, Washington, DC (2009)
64. Tun, T.T., Heymans, P.: Concerns and their separation in feature diagram languages - an informal survey. In: Workshop on Scalable Modelling Techniques for Software Product Lines (SCALE@SPLC'09), pp. 107–110, 2009
65. Weiser, M.: Program slicing. In: Proceedings of the International Conference on Software Engineering (ICSE '81), pp. 439–449. IEEE Computer Society, Washington, DC (1981)
66. Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12), pp. 58–68. IEEE Computer Society, Washington, DC (2012)
67. Zhao, H., Zhang, W., Mei, H.: Multi-view based customization of feature models. J. Front. Comput. Sci. Tech. **2**(3), 260–273 (2008)

A Survey of Feature Location Techniques

Julia Rubin and Marsha Chechik

Abstract Feature location techniques aim at locating software artifacts that implement a specific program functionality, a.k.a. *a feature*. These techniques support developers during various activities such as software maintenance, aspect- or feature-oriented refactoring, and others. For example, detecting artifacts that correspond to product line features can assist the transition from unstructured to systematic reuse approaches promoted by software product line engineering (SPLE). Managing features, as well as the traceability between these features and the artifacts that implement them, is an essential task of the SPLE domain engineering phase, during which the product line resources are specified, designed, and implemented. In this chapter, we provide an overview of existing feature location techniques. We describe their implementation strategies and exemplify the techniques on a realistic use-case. We also discuss their properties, strengths, and weaknesses and provide guidelines that can be used by practitioners when deciding which feature location technique to choose. Our survey shows that none of the existing feature location techniques are designed to consider families of related products and only treat different products of a product line as individual, unrelated entities. We thus discuss possible directions for leveraging SPLE architectures in order to improve the feature location process.

Keywords Feature location • Software maintenance • Software product lines

J. Rubin (✉)
IBM Research, Haifa, Israel and University of Toronto, Canada
e-mail: mjulia@il.ibm.com

M. Chechik
University of Toronto, Canada
e-mail: chechik@cs.toronto.edu

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_2, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

Software product line engineering (SPLE) techniques [10, 25] capitalize on identifying and managing *common* and *variable* product line features across a product portfolio. SPLE promotes *systematic* software reuse by leveraging the knowledge about the set of available features, relationships among the features and relationships between the features and software artifacts that implement them. However, in reality, software families—collections of related software products—often emerge ad hoc, from experiences in successfully addressed markets with similar, yet not identical needs. Since it is difficult to foresee these needs a priori and hence to design a software product line upfront, software developers often create new products by using one or more of the available technology-driven software reuse techniques such as duplication (the "clone-and-own" approach), source control branching, preprocessor directives, and more.

Essential steps for unfolding the complexity of existing implementations and assisting their transformation to systematic SPLE reuse approaches include identification of implemented features and detection of software artifacts that realize those features. While the set of available features in many cases is specified by the product documentation and reports, the relationship between the features and their corresponding implementation is rarely documented. Identification of such relationships is the main goal of feature location techniques.

Rajlich and Chen [8] represent a feature (a.k.a. a *concept*) as a triple consisting of a *name*, *intension*, and *extension*. The name is the label that identifies the feature; intension explains the meaning of the feature; and extension is a set of artifacts that realize the feature. *Location: intension* $\rightarrow$ *extension* is identified by the authors as one of the six fundamental program comprehension processes. Its application to features is the subject of this survey.

In the remainder of the chapter, we illustrate the surveyed concepts using a problem of locating the *automatic save file* feature, previously studied in [32], in the code of the Freemind[1] open source mind-mapping tool. A snippet of Freemind's call graph is shown in Fig. 1. Shaded elements in the graph contribute to the implementation of the *automatic save file* feature—they are the feature *extension* which we want to locate. Feature *intension* can be given, for example, by the natural language query "*automatic save file*," describing the feature.[2]

The feature is mainly implemented by two methods of the `MindMapMapModel` subclass `doAutomaticSave`: the constructor and the method `run` (elements #1 and #2). `doAutomaticSave` class is initiated by the `MindMapMapModel`'s constructor (element #4), as shown in Fig. 2. The constructor assigns values to several configuration parameters related to the *automatic save file* function and then registers the `doAutomaticSave` class on the scheduling queue. This initiates the

[1] http://freemind.sourceforge.net.

[2] We denote features by *italic font*, place natural language queries "*in quotes*," and denote code elements by a `monospaced font`.

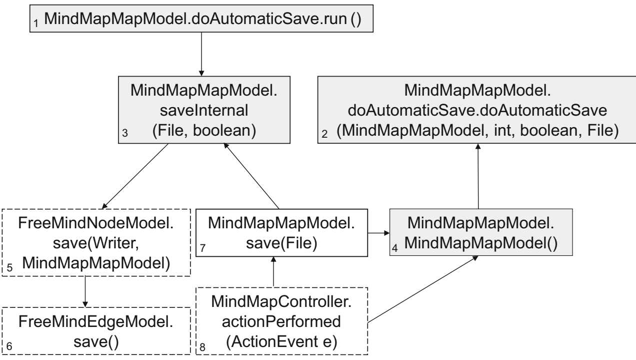


Fig. 1 The *automatic save file* call graph snippet

```
public MindMapMapModel( MindMapNodeModel root,
    ⋮                    FreeMindMain frame ) {
  // automatic save:
  timerForAutomaticSaving = new Timer();
  int delay = Integer.parseInt(getFrame().
    getProperty("time_for_automatic_save"));
  int numberOfTempFiles = Integer.parseInt(getFrame().
    getProperty("number_of_different_files_for_automatic_save"));
  boolean filesShouldBeDeletedAfterShutdown = Tools.
    safeEquals(getFrame().
    getProperty("delete_automatic_save_at_exit"),"true");
  String path = getFrame().getProperty("path_to_automatic_saves");
    ⋮
  timerForAutomaticSaving.schedule(new doAutomaticSave(
    this, numberOfTempFiles,
    filesShouldBeDeletedAfterShutdown, dirToStore),
    delay, delay);
  );
}
```

Fig. 2 The `MindMapMapModel` code snippet

class's `run` method (element #1) which subsequently calls the `saveInternal` method (element #3) responsible for performing the *save* operation.

Obviously, not all program methods contribute to the *automatic save file* feature. For example, element #3 also initiates a call to `FreeMindNodeModel`'s `save( Writer, MindMapMapModel)` method (element #5), which, in turn, calls element #6–`save(Writer, MindMapMapModel)`. Both of these methods are irrelevant to the specifics of the *automatic save file* implementation. Element #3 itself is called by element #7 (`MindMapMapMode`'s `save(File)` method), which is called by element #8 (`MindMapController`'s `actionPerformed(ActionEvent)`). These methods are also not relevant to the feature implementation because they handle a *user-triggered save* operation instead of *automatic save*. In fact, element #8 initiates calls to an additional 24 methods, all of which are irrelevant to the implementation of the feature. In Fig. 1, irrelevant methods are not shaded.

While all feature location approaches share the same goal—establishing traceability between a specific feature of interest specified by the user (feature intension) and the artifacts implementing that feature (feature extension), they differ substantially in the underlying design principles, as well as in assumptions they make on their input (representation of the intension). In this chapter, we provide an in-depth description of 24 existing feature location techniques and their underlying technologies. We exemplify them on a small but realistic program snippet of the Freemind software introduced above and discuss criteria for choosing a feature location technique based on the qualities of the input program. We also assess the techniques by the amount of required user interaction.

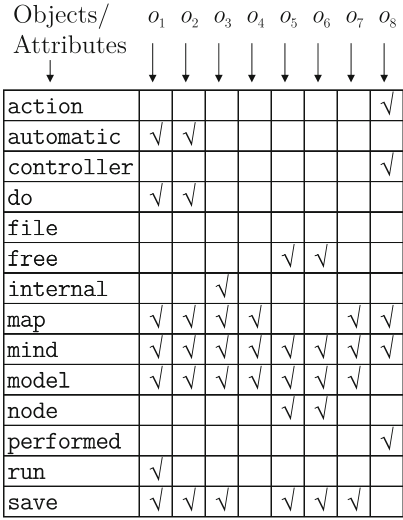
Our specific interest is in applying feature location techniques in the context of software families where a feature can be implemented by multiple products. However, none of the existing techniques explicitly consider collections of related products when performing feature location: the techniques are rather applied to these products as if these are unrelated, singular entities. Thus, another contribution of our work is a discussion of research directions towards a more efficient feature location, taking advantage of existing families of related products (see Sect. 6).

A systematic literature survey of 89 articles related to feature location is available in [11]. That survey provides a broad overview of existing feature definition and location techniques, techniques for feature representation and visualization, available tools and performed user studies. The purpose of that work is organizing, classifying and structuring existing work in the field and discussing open issues and future directions. Even though 22 out of the 24 techniques surveyed here are covered by [11], our work has a complementary nature. We focus only on *automated feature location* techniques while providing insights about the implementation details, exemplifying the approaches and discussing how to select one in real-life settings. The intended audience of our survey is practitioners aiming to apply a feature location technique for establishing traceability between the features of their products and the implementation of these features. As such, these practitioners have to understand the implementation details and properties of the available approaches in order to choose one that fits their needs.

The rest of the chapter is organized as follows. In Sect. 2, we start by introducing basic technologies used by several feature location techniques. Section 3 introduces the classification that we use for the surveyed feature location techniques. A detailed description of the techniques themselves is provided in Sect. 4. We discuss criteria used when selecting a feature location technique in Sect. 5. Section 6 concludes our survey and presents directions for possible future work on feature location in the context of SPLE.

2 Basic Underlying Technologies

In this section, we introduce basic technologies commonly used by feature location techniques, describe each technology, and demonstrate it on the example in Sect. 1.



Objects/ Attributes	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
`action`								√
`automatic`	√	√						
`controller`								√
`do`	√	√						
`file`								
`free`					√	√		
`internal`			√					
`map`	√	√	√	√			√	√
`mind`	√	√	√	√	√	√	√	√
`model`	√	√	√	√	√	√	√	
`node`					√	√		
`performed`								√
`run`	√							
`save`	√	√	√		√	√	√	

Fig. 3 Formal context for the example in Fig. 1. Objects are method names, attributes are tokens of the names

2.1 *Formal Concept Analysis*

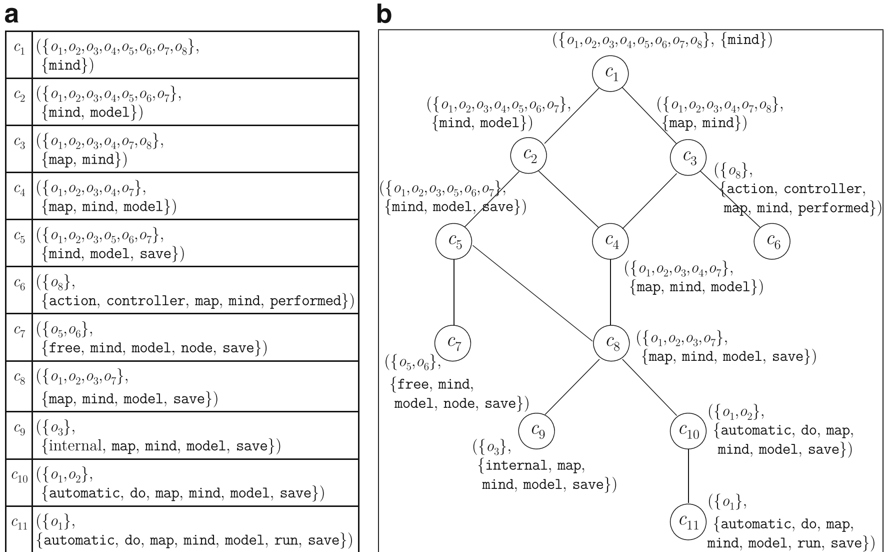
Formal Concept Analysis (FCA) [16] is a branch of mathematical lattice theory that provides means to identify meaningful *groupings* of objects that share common attributes. Groupings are identified by analyzing binary relations between the set of all objects and all attributes. FCA also provides a theoretical model to analyze hierarchies of these identified groupings.

The main goal of FCA is to define a *concept* as a unit of two parts: *extension* and *intension*.[3] The extension of a concept covers all the objects that belong to the concept, while the intension comprises all the attributes, which are shared by all the objects under consideration. In order to apply FCA, the *formal context* of objects and their respective attributes is necessary. The formal context is an incidence table indicating which attributes are possessed by each object. An example of such a table is shown in Fig. 3, where objects are names of methods in Fig. 1 and attributes are individual words obtained by tokenizing and lowercasing these names. For example, object o_1 corresponds to element #1 in Fig. 1 and is tokenized to attributes `automatic`, `do`, `map`, `mind`, `model`, `run`, `save`, which are "checked" in Fig. 3.

From the formal context, FCA generates a set of concepts where every concept is a maximal collection of objects that possess common attributes. Figure 3a shows all concepts generated for the formal context in Fig. 3.

Formally, given a set of objects O, a set of attributes A, and a binary relationship between objects and attributes R, the *set of common attributes* is defined as $\sigma(O) = \{a \in A \mid (o, a) \in R \ \forall o \in O\}$. Analogously, *the set of common objects* is defined as

[3]These are not to be confused with the extension and intension of a feature.



a

c_1	$(\{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8\}$, {mind})
c_2	$(\{o_1, o_2, o_3, o_4, o_5, o_6, o_7\}$, {mind, model})
c_3	$(\{o_1, o_2, o_3, o_4, o_7, o_8\}$, {map, mind})
c_4	$(\{o_1, o_2, o_3, o_4, o_7\}$, {map, mind, model})
c_5	$(\{o_1, o_2, o_3, o_5, o_6, o_7\}$, {mind, model, save})
c_6	$(\{o_8\}$, {action, controller, map, mind, performed})
c_7	$(\{o_5, o_6\}$, {free, mind, model, node, save})
c_8	$(\{o_1, o_2, o_3, o_7\}$, {map, mind, model, save})
c_9	$(\{o_3\}$, {internal, map, mind, model, save})
c_{10}	$(\{o_1, o_2\}$, {automatic, do, map, mind, model, save})
c_{11}	$(\{o_1\}$, {automatic, do, map, mind, model, run, save})

Fig. 4 Concepts and the corresponding concept lattice generated for the formal context in Fig. 3. (**a**) Concept, (**b**) Concept lattice

$\rho(O) = \{o \in O \mid (o, a) \in R \ \forall a \in A\}$. For example, for the relationship R encoded in Fig. 3, $\sigma(o_4) = \{\texttt{map}, \texttt{mind}, \texttt{model}\}$ and $\rho(\texttt{automatic}, \texttt{do}) = \{o_1, o_2\}$.

A *concept* is a pair of sets (O, A) such that $A = \rho(O)$ and $O = \sigma(A)$. O is considered to be the *extension* of the concept and A is the *intension* of the concept. The set of all concepts of a given formal context forms a partial order via the superconcept-subconcept ordering $\leq$: $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2$, or, dually, $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_2 \subseteq A_1$.

The set of all concepts of a given formal context and the partial order $\leq$ form a *concept lattice*, as shown in Fig. 4b. In our example, this lattice represents a taxonomy of tokens used for naming the methods—from the most generic used by all methods (the root element c_1, which represents the term `mind` used in all names) to the more specific names depicted as leaves (e.g., c_6 which represents unique terms `action`, `controller` and `performed` used in the name of element #8).

2.2 *Latent Semantic Indexing*

Latent semantic indexing (LSI) [21] is an automatic mathematical/statistical technique that analyzes the relationships between queries and passages in large bodies of text. It constructs vector representations of both a user query and a corpus of text documents by encoding them as a *term-by-document co-occurrence matrix*. Each row in the matrix stands for a unique word, and each column stands for a text passage or a query. Each cell contains the frequency with which the word of its row appears in the passage denoted by its column.

Fig. 5 Term-by-document co-occurrence matrix for the example in Fig. 1. Documents are method names, terms are tokens of the names and the query

Documents/Terms	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	q
action	0	0	0	0	0	0	0	1	0
automatic	1	2	0	0	0	0	0	0	1
controller	0	0	0	0	0	0	0	1	0
do	1	2	0	0	0	0	0	0	0
file	0	0	0	0	0	0	0	0	1
free	0	0	0	0	1	1	0	0	0
internal	0	0	1	0	0	0	0	0	0
map	2	2	2	4	0	0	2	1	0
mind	1	1	1	2	1	1	1	1	0
model	1	1	1	2	1	1	1	0	0
node	0	0	0	0	1	1	0	0	0
performed	0	0	0	0	0	0	0	1	0
run	1	0	0	0	0	0	0	0	0
save	1	2	1	0	1	1	1	0	1

(A = the matrix of columns $d_1 \dots d_8$; q = the query column)

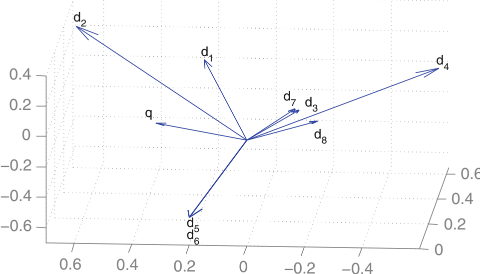


Fig. 6 Vectorial representation of the documents and the query in Fig. 5

Figure 5 shows such an encoding for the example in Fig. 1, where "documents" are method names, the query "*automatic save file*" is given by the user, and the set of all terms is obtained by tokenizing, lowercasing, and alphabetically ordering strings of both the documents and the query. In Fig. 5, matrix A represents the encoding of the documents and matrix q represents the encoding of the query. Vector representations of the documents and the query are obtained by normalizing and decomposing the *term-by-document co-occurrence matrix* using a matrix factorization technique called *singular value decomposition* [21]. Figure 6 shows the vector representation of the documents $d_1 \dots d_8$ and the query q in Fig. 5 in a three-dimensional space.

The similarity between a document and a query is typically measured by the cosine between their corresponding vectors. The similarity increases as the vectors point "in the same general direction," i.e., as more terms are shared between the documents. For the example in Fig. 6, document d_2 is the most similar to the query, while d_8 is the least similar. The exact similarity measures between the document and the query, as calculated by LSI, are summarized in Table 1. It is common to consider documents with positive similarity values as related to the query of interest (i.e., d_1, d_2, d_5 and d_6 in our example), while those with negative similarity values (i.e., d_3, d_4, d_7 and d_8)—as unrelated.

Table 1 Similarity of the documents and the query in Fig. 5 as calculated by LSI

d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
0.6319	0.8897	−0.2034	−0.5491	0.2099	0.2099	−0.1739	−0.6852

2.3 Term Frequency: Inverse Document Frequency Metric

Term frequency—inverse document frequency (*tf-idf*) is a statistical measure often used by IR techniques to evaluate how important a term is to a specific document in the context of a set of documents (*corpus*). Intuitively, the more frequently a term occurs in the document, the more relevant the document is to the term. That is, the relevance of a specific document d to a term t is measured by *document frequency* ($tf(t,d)$). For the example in Fig. 5 where "documents" are names of methods in Fig. 1, the term `save` appears twice in d_2, thus $tf(\texttt{save}, d_2) = 2$.

The drawback of term frequency is that uninformative terms appearing throughout the set D of all documents can distract from less frequent, but relevant, terms. Intuitively, the more documents include a term, the less this term discriminates between documents. The *inverse document frequency*, *idf(t)*, is then calculated as follows: $idf(t) = \log(\frac{|D|}{|\{d \in D \mid t \in d\}|})$. The *tf-idf* score of a term w.r.t. a document is calculated by multiplying its *tf* and *idf* scores: *tf-idf(t,d)* $= tf(t,d) \times idf(t)$. In our example, $idf(\texttt{save}) = \log(\frac{8}{6}) = 0.12$ and $tf\text{-}idf(\texttt{save}, d_2) = 2 \times 0.12 = 0.24$.

Given a query which contains multiple terms, the *tf-idf* score of a document with respect to the query is commonly calculated by adding *tf-idf* scores of all query terms. For example, the *tf-idf* score of d_2 with respect to the query "*automatic save file*" is 1.44, while d_3 score with respect to the same query is 0.12.

2.4 Hyper-link Induced Topic Search

Hyper-link Induced Topic Search (HITS) is a page ranking algorithm for Web mining introduced by Kleinberg [19]. The algorithm considers two forms of web pages—*hubs* (pages which act as resource lists) and *authorities* (pages with important content). A good hub points to many authoritative pages whereas a good authority page is pointed to by many good hub pages.

The HITS algorithm operates on a directed graph, whose nodes represent pages and whose edges correspond to links. Authority and hub scores for each page p (denoted by A_p and H_p, respectively) are defined in terms of each other: $A_p = \sum_{\{q \mid q \text{ points to } p\}} H_q$ and $H_p = \sum_{\{q \mid p \text{ points to } q\}} A_q$. The algorithm initializes hub and authority scores of each page to 1 and performs a series of iterations. Each iteration calculates and normalizes the hub (authority) value of each page. It does so by dividing the value by the square root of the sum of squares of all hub (authority) values for the pages that it points to (pointed by). The algorithm stops when it reaches a fixpoint or a maximum number of iterations.

When applied to code, HITS scores methods in a program based on their "strength" as hubs—aggregators of functionality, i.e., methods that call many others, and authorities—those that implement some functionality without aggregation. For the example in Fig. 1, elements #2 and #6 are authorities as they do not call any other methods and thus their hub score is 0. Elements #1 and #8 are hubs as they are not called by other methods. Thus, their authority score is 0. Elements #3 and #4 get a higher authority score than other elements as they are called by two methods each, while elements #7 and #8 get a higher hub score than the rest as they call two methods each.

3 Classification and Methodology

In this section, we discuss the classification of feature location techniques that we use for organizing our survey. We also discuss main properties that we highlight for each technique.

Primarily, feature location techniques can be divided into *dynamic* which collect information about a program at runtime and *static* which do not involve program execution. The techniques also differ in the way they assist the user in the process of interpreting the produced results. Some only present an (unsorted) list of artifacts considered relevant to the feature of interest; we refer to these as *plain output* techniques. Others provide additional information about the output elements, such as their relative ranking based on the perceived relevance to the feature of interest or automated and guided output exploration process which suggests the order and the number of output elements to consider; we refer to these as *guided output* techniques. Figure 7 presents the surveyed techniques, dependencies between them and their primary categorization.

Feature location approaches can rely on *program dependence analysis (PDA)* that leverages static dependencies between program elements; *information retrieval (IR)* techniques—in particular, *LSI*, *tf-idf* and others, that leverage information embedded in program identifier names and comments; *change set analysis* that leverages historical information and more. While dynamic approaches collect precise information about the program execution, they are safe only with respect to the input that was actually considered during runtime to gather the information, and generalizing from this data may not be safe [20]. In addition, while generally a *feature* is a realization of a system requirement—either functional or non-functional—executable test-cases or scenarios can exhibit only *functional* requirements of the system that are visible at the user level. Thus, dynamic feature location techniques can detect only functional features. On the other hand, static approaches can locate any type of feature and yield safe information, but because many interesting properties of programs are statically undecidable in general, static analysis is bound to approximate solutions which may be too imprecise in practice. Dynamic analysis yields "under-approximation" and thus might suffer

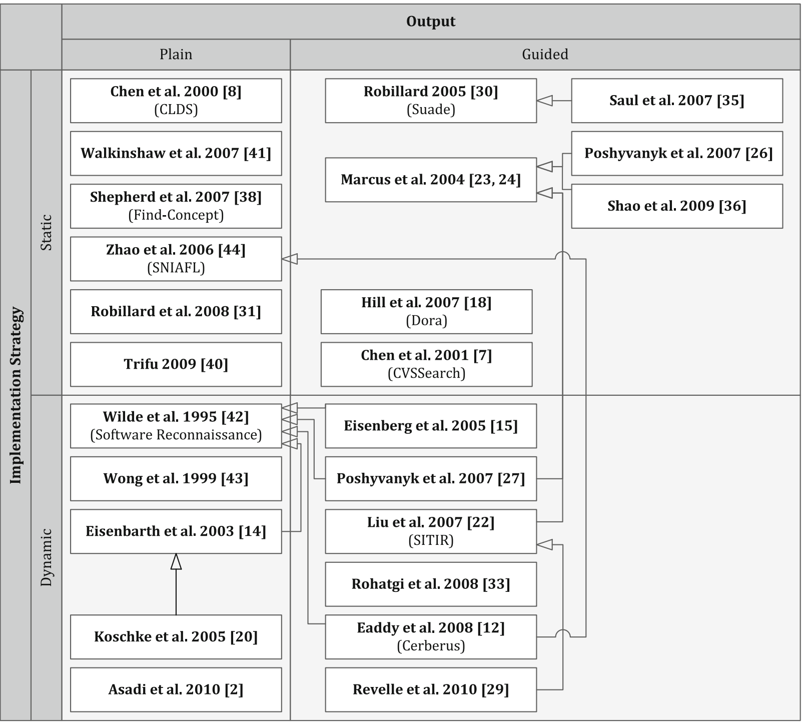


Fig. 7 Surveyed techniques and their categorization

from many false-negative results while static analysis yields "over-approximation" and thus might have many false-positives. In order to find a middle ground, *hybrid* approaches combine several techniques.

Based on the chosen implementation technique, the analyzed program can be represented as a *program dependence graph (PDG)*, a set of text documents representing software elements, an instrumented executable that is used by dynamic techniques and more. Figure 8 provides detailed information about each of the surveyed techniques, listing its underlying technology, the chosen program representation, the type of user input, as well as the amount of required user interaction, ranging from low (denoted by "+") to high (denoted by "+ + +").

4 Feature Location Techniques

In this section, we describe automated feature location techniques from the literature. As discussed in Sect. 1, we focus on those techniques that assist the user with feature *location* rather than feature definition or visualization. Static approaches

		Technique	Underlying Technology	Program Representation	Input	User Interaction
Static	Plain	**Chen et al. [8]** (CLDS)	PDA	PDG	method or global variable	+++
		Walkinshaw at al. [41]	PDA	call graph	two sets of methods	+
		Shepherd et al. [38] (Find-concept)	PDA, NLP	AOIG	query	++
		Zhao et al. [44] (SNIAFL)	TF-IDF, vector space model, PDA	BRCG	set of queries	+
		Robillard et al. [31]	clustering algorithms	change history transactions	set of elements	+
		Trifu [40]	PDA	concern graph	set of variables	+++
	Guided	**Robillard [30]** (Suade)	PDA	PDG	set of methods and global variables	++
		Saul et al. [35]	PDA, web-mining algorithm	call graph	method	++
		Marcus et al. [23, 24]	LSI	text docs for software elements	query	+
		Poshyvanyk et al. [26]	LSI, FCA on retrieved docs	text docs for software elements	query	+
		Shao et al. [36]	LSI, PDA	call graph, text docs for software elements	query	+
		Hill et al. [18] (Dora)	PDA, TF-IDF	call graph, text docs for software elements	method, query	+
		Chen et al. [7] (CVSSearch)	textual search	lines of code, CVS comments	query	+
Dynamic	Plain	**Wilde et al. [42]** (Sw. Reconnaissance)	trace analysis	executable	set of test cases	+++
		Wong et al. [43]	trace analysis	executable	set of test cases	+++
		Eisenbarth et al. [14]	trace analysis (FCA), PDA	executable, PDG	set of scenarios	+++
		Koschke et al. [20]	trace analysis (FCA), PDA	executable, statement dependency graph	set of scenarios	+++
		Asadi et al. [2]	trace analysis, LSI, genetic optimization	executable, text docs for methods	set of scenarios	++
	Guided	**Eisenberg et al. [15]**	trace analysis	executable	set of test cases	++
		Poshyvanyk et al. [27]	trace analysis, LSI	executable, text docs for methods	set of scenarios, query	+++
		Liu et al. [22] (SITIR)	trace analysis, LSI	executable, text docs for methods	scenario, query	+
		Rohatgi et al. [33]	trace analysis, impact analysis	executable, class dependency graph	set of scenarios	++
		Eaddy et al. [12] (Cerberus)	PDA, trace analysis, TF-IDF, vector space model	PDG, executable, text docs for software elements	set of queries, set of scenarios	+++
		Revelle et al. [29]	trace analysis, LSI, web-mining algorithm	executable, text docs for methods	scenario, query	+

Fig. 8 Underlying technology, program representation, and input type of the surveyed techniques

(those that do no require program execution) are described in Sect. 4.1; dynamic are in Sect. 4.2.

4.1 *Static Feature Location Techniques*

In this section, we describe techniques that rely on static program analysis for locating features in the source code.

4.1.1 Plain Output

Chen et al. [8] present one of the earliest static computer-assisted feature location approaches based on *PDA*. The analyzed program is represented as a *PDG* whose nodes are methods or global variables and edges are method invocations or data access links (the paper refers to the PDG as the abstract system dependence graph). Given an initial element of interest—either a function or a global variable—the approach allows the user to explore the PDG interactively, node-by-node, while storing visited nodes in a *search graph*. The user decides whether the visited node is related to the feature and marks related nodes as such. The process stops when the user is satisfied with the set of found nodes, and outputs the set of relevant nodes aggregated in the search graph. For the example in Fig. 1, the system generates the depicted call graph from the source code and interactively guides the user through its explanation. The technique relies on extensive user interaction (denoted by "+++" in Fig. 8), and thus provides the user with "intelligent assistance" [6] rather than being a heuristic-based technique aiming to determine relevant program elements automatically.

Walkinshaw et al. [41] provide additional automation to the feature location process based on PDA. The analyzed program is represented as a *call graph*—a subgraph of PDG containing only methods and method invocations. As input, the system accepts two sets of methods: *landmark*—thought to be essential for the implementation of the feature, and *barrier*—thought to be irrelevant to the implementation. For the example in Fig. 1, landmark methods could be elements #1 and #2, while barrier methods—#5 and #7. The system computes a *hammock graph* which contains all of the direct paths between the landmarks. That is, a method call belongs to the hammock graphs only if it is on a direct path between a pair of landmark methods. Additional potentially relevant methods are added to the graph using intra-procedural *backward slicing* [39] (with the call sites that spawn calls in the hammock graph as slicing criteria). Since slicing tends to produce graphs that are too large for practical purposes, barrier methods are used to eliminate irrelevant sections of the graph: all incoming and outgoing call graph edges of barrier methods are removed, and thus these are not traversed during the slice computation. The approach outputs all elements of the resulting graph as relevant to the feature.

In our example in Fig. 1, no direct call paths exist between elements #1 and #2; thus, the approach is unable to find additional relevant elements under the given input. The technique is largely automated and does not require extensive user interaction (denoted by "+" in Fig. 8) other than providing and possibly refining the input sets of methods.

Shepherd et al. [38] attempt to locate action-oriented concepts in object-oriented programs using domain knowledge embedded in the source code through identifier names (methods and local variables) and comments. It relies on the assumption that verbs in object-oriented programs correspond to methods, whereas nouns correspond to objects.

The analyzed program is represented as an *action-oriented identifier graph model (AOIG)* [37] where the actions (i.e., verbs) are supplemented with direct

objects of each action (i.e., objects on which the verb acts). For example, the verb `save` in Fig. 1 can act on different objects in a single program, such as `MindMapMapModel` and `MindMapNodeModel`; these are the direct objects of `save`. An AOIG representation of a program contains four kinds of nodes: *verb nodes*, one for each distinct verb in the program; *direct object (DO) nodes*, one for each unique direct object in the program; *verb-DO nodes*, one for each verb-DO pair identified in the program (a verb or a direct object can be part of several verb-DO pairs); and *use nodes*, one for each occurrence of a verb-DO pair in comments or source code of the program. An AOIG has two kinds of edges: *pairing edges* connecting each verb or DO node to verb-DO pairs that use them, and *use edges* connecting each verb-DO pair to all of its use nodes.

As an input, the user formulates a query describing the feature of interest and *decomposes* it into a set of pairs (verb, direct object). The technique helps the user to refine the input query by collecting verbs and direct objects that are similar (i.e., different forms of words, and synonyms) to the input verbs and direct objects, respectively, as well as words collocated with those in the query, based on the verb-DO pairs of the program AOIG. For example, `MindMapMapModel` is collocated with `MindMapNodeModel` in verb-DO pairs for the verb `save`. The collected terms are ranked by their "closeness" to the words in the query based on the frequency of collocation with the words in the query and on configurable weight given to synonyms. Ten best-ranked terms are presented to the user. The system then recommends that the user augment the query with these terms as well as with program methods that match the current query.

Once the user is satisfied with the query, the system searches the AOIG for all verb-DO pairs that contain the words of the query. It extracts all methods where the found pairs are used and applies PDA to detect call relationships between the extracted methods. The system then generates the *result graph* in which nodes represent detected methods and edges represent identified structural relationships between them. The graph is returned to the user.

For our example in Fig. 1, the input query (`doAutomaticSave`, `MindMapMapModel`) might get expanded by the user with the terms `save` and `saveInternal`, because they are collocated with `MindMapMapModel`. Then, the system outputs elements #1 through #4 and #7, together with the corresponding call graph fragment. The technique requires a fair amount of user interaction to construct and refine the input query, and thus is marked with "++" in Fig. 8.

Zhao et al. [44] accept a set of feature descriptions as input and focus on locating the *specific* and the *relevant* functions of each feature using PDA and IR technologies. The specific functions of a feature are those definitely used to implement it but are not used by other features. The relevant functions of a feature are those involved in the implementation of the feature. Obviously, the specific function set is a subset of the relevant function set for every feature.

The analyzed program is represented as a *Branch-Reserving Call Graph (BRCG)* [28]—an expansion of the call graph with branching and sequential information, which is used to construct the pseudo execution traces for each feature. Each node in the BRCG is a function, a branch, or a return statement. Loops are regarded as

two branch statements: one going through the loop body and the other one exiting immediately. The nodes are related either sequentially, for statements executed one after another, or conditionally, for alternative outcomes of a branch.

The system receives a paragraph of text as a description of each feature. The text can be obtained from the requirements documentation or be provided by a domain expert. It transforms each feature description into a set of index terms (considering only nouns and verbs and using their normalized form). These will be used as documents. The system then extracts the names of each method and its parameters, separating identifiers using known coding styles (e.g., using the underline "_" to separate words) and transforms them into index terms. These will be used as queries.

To reveal the connections between features and functions, documents (feature descriptions) are ranked for each query (function) using the vector space models [3, pp. 27–30]—a technique which, similar to LSI, treats queries and documents as vectors constructed by the index terms. Unlike LSI, the weights of index term in documents and queries are calculated using the *tf-idf* metric (see Sect. 2.3) between the term and the document or query, respectively. For the example in Fig. 1, *automatic save file* could be a document while "*mind map model do automatic save*" could be a query corresponding to the element #2. For the vector space model, the weight of the term `save` in the query is 0.24, as calculated in Sect. 2.3. Note that LSI calculates this weight as being 2 (see the value of the term `save` in the column that corresponds to d_2 in Fig. 5).

Similarity between a document and a query is computed as a cosine of the angle between their corresponding vectors, as for LSI. For each document (feature), the system creates a sorted list of queries (functions), ranked by their similarity degrees and identifies a pair of functions with the largest difference between scores. All functions before this pair, called *a division point*, are considered *initial specific functions* to the feature. In our example, these are elements #1 and #2.

Next, the system analyzes the program's BRCG and filters out all branches that do not contain any of the initial specific functions of the feature, because those are likely not relevant; all remaining functions are marked as *relevant*. Functions relevant to exactly one feature are marked as *specific* to that feature.

The system also builds pseudo-execution traces for each feature by traversing the pruned BRCG and returns those to the user. For our example in Fig. 1, BRCG is rooted in element #8. Since there is no direct call to element #1 (the call is performed via an event queue—see the last statement in Fig. 2), the technique returns only those branches that contain element #2, that is, elements #8, #7, and #4. The technique requires no user interaction besides the definition and the refinement of the input feature descriptions, as reflected by "+" in Fig. 8.

Robillard et al. [31] propose searching the change history (*change transactions*) of a software system to identify clusters of program elements related to a task. The analyzed program is represented as a set of program elements such as fields and methods, as well as change history transactions that capture modifications of these elements. The system considers all available transactions and filters out those with more than 20 or fewer than four elements. The thresholds are set empirically: experiments revealed that large transactions generate overly large clusters that

would require developers to spend an unreasonable amount of effort to study, while small transactions cannot be clustered efficiently. The system then clusters the remaining transactions based on the number of overlapping elements using a standard clustering algorithm.

Next, given a small set of elements related to a feature of interest (usually two or three), the system extracts clusters containing all input elements and removes those satisfying the following conditions:

1. An input element appears in at least 3% of the transactions of the cluster. The rationale is that querying the change history for elements that are being continuously modified (and thus are central or critical elements to the entire system) returns too many recommendations to be useful.
2. The degree of overlap between elements that correspond to the transactions in a cluster is lower than 0.6. The rationale is that these clusters do not represent changes that are associated with a high-level concept.
3. The number of transactions in a cluster is less than 3. The rationale is to avoid returning results that are single transactions or very small groups of transactions which may have been spontaneously clustered. However, using a value higher than three as a threshold produces too few recommendations to be useful.

All elements of the resulting clusters are returned to the user. The technique requires no user interaction besides the definition and the refinement of the input elements, as reflected by "+" in Fig. 8.

Unfortunately, the evaluation of the proposed technique which is included in the paper shows that the benefits of using change clusters are relatively small: the analysis of almost 12 years of software change data for a total of seven different open-source systems showed that fewer than 12% of the studied changes could have benefited from finding elements relevant to the change using change clusters.

Trifu [40] proposes an approach that uses *static dataflow information* to determine the *concern skeleton*—a data-oriented abstraction of a feature. The analyzed program is represented as a *concern graph* whose nodes are variables found in the source code and whose edges are either *dataflow relationships* that capture value transfer between variables or *inheritance relationships* that insure consistent handling of variables defined in polymorphic methods. A path between two variables indicates that the start variable is used to derive the value of the end variable.

The approach treats a feature as an implementation of functionality needed to produce a given set of related values. It receives as input a set of variables that store key results produced by the feature of interest—*information sinks*—and computes a *concern skeleton* which contains all variables in the concern graph that have a path to one of the information sinks. The approach can be optionally provided with an additional set of input variables—*information sources*—that act as cutting points for the incoming paths leading to an information sink. That is, the computed concern skeleton includes only portions of the paths from the given information sources to the given information sinks. The computed concern skeleton is returned to the user.

The approach provides some help in identifying the input set of information sinks by computing a *reduced concern graph* in which certain variables are filtered out (e.g., those that have no incident edges in the concern graph). Still, identifying information sinks is not a trivial task which involves semantic knowledge about what the system does. Also, the user has to do the mapping from variables of the resulting concern skeleton to program statements that use them. Thus, the technique relies on extensive user interaction, as indicated by "+ + +" in Fig. 8.

4.1.2 Guided Output

Robillard [30] leverages static program dependence analysis to find elements that are related to an initial *set of interest* provided by the user. The analyzed program is represented as a PDG whose nodes are functions or data fields and edges are function calls or data access links. Given an input *set of interest*—a set of functions and data fields that the user considers relevant to the feature of interest, the system explores their neighbors in the dependency graph and scores them based on their *specificity*—an element is specific if it relates to few other elements, and *reinforcement*—an element is reinforced if it is related to other elements of interest. For the example in Fig. 1, if the initial set of interest contains elements #3 and #4, reinforcement of element number #7 is high as two of its three connections are to elements of interest. Reinforcement of element #1 is even higher, as its sole connection leads to an element of interest. Yet, specificity of element #7 is lower than that of element #1 since the former is connected to three elements whereas the latter—just to one.

The set of all elements related to those in the initial set of interest is scored and returned to the user as a sorted *suggestion set*. The user browses the result, adds additional elements to the set of interest and reiterates. The amount of the required user interaction in this approach is moderate, as indicated by "++" in Fig. 8: the technique itself only browses the direct neighbors of the elements in the input *set of interest* while the user is expected to extend this set interactively, using the results generated by the previous step.

Saul et al. [35] build on Robillard's technique [30]) and introduce additional heuristics for scoring program methods. The proposed approach consists of two phases: in the first, a set of potentially relevant methods is calculated for an input method of interest. These are the union of caller and callee methods ("parents" and "children"), methods called by the caller functions ("siblings") and methods that call the callee methods ("spouses"). For example, for the element #4 in Fig. 1, elements #2, #3, #7, and #8 are potentially relevant.

The calculated set of potentially relevant methods is then scored using the HITS web mining algorithm (see Sect. 2.4) based on their "strength" as *hubs* (methods that aggregate functionality, i.e., call many other methods) or *authorities* (methods that largely implement functionality without aggregating). The calculated authority score is used to rank the results returned by the algorithm. That is, a method gets a high score if it is called by many high-scored hub methods. In our example, element

#7 has a lower score than #4, because the former is called only by method #8 which is a low-scored hub method as it calls only one method. Element #4 has a higher score because (1) it is called by both elements #7 and #8, and (2) element #7 has a higher hub score as it calls two methods rather than one.

Similar to [30], the technique requires a moderate amount of user interaction, as indicated by "++" in Fig. 8.

Marcus et al. [23,24] introduce one of the first approaches for using IR techniques for feature location. The approach is based on using domain knowledge embedded in the source code through identifier names and internal comments.

The analyzed program is represented as a set of text documents describing software elements such as methods or data type declarations. To create this set of documents (corpus), the system extracts identifiers from the source code and comments, and separates the identifiers using known code styles (e.g., the use of underline "_" to separate words). Each software element is described by a separate document containing the extracted identifiers and translated to LSI space vectors (see Sect. 2.2) using identifiers as terms.

Given a natural language query containing one or more words, identifiers from the source code, a phrase or even short paragraphs formulated by the user to identify a feature of interest,[4] the system converts it into a document in LSI space, and uses the similarity measure between the query and documents of the corpus in order to identify the documents most relevant to the query.

In order to determine how many documents the user should inspect, the approach partitions the search space based on the similarity measure: each partition at step $i + 1$ is made up of documents that are closer than a given threshold α to the most relevant document found by the user in the previous step i. The user inspects the suggested partition and decides which documents are part of the concept. The algorithm terminates once the user finds no additional relevant documents in the currently inspected partition and outputs a set of documents that were found relevant by the user, ranked by the similarity measure to the input query.

For the example in Fig. 1, assume that similarities between documents and a query are calculated as specified in Sect. 2.2 and summarized in Table 1. That is, only terms from method names (and not from method bodies) are used. Under this setting, if α is set to 0.3, the first partition will contain only document d_2 and the second—only d_1. No other document is within 0.3 of d_1 and thus the algorithm will terminate and output d_1 and d_2.

The technique requires no user interaction besides the definition and the refinement of the input query, and thus is marked with "+" in Fig. 8.

Poshyvanyk et al. [26] extend the work of Markus et al. [23, 24] with formal concept analysis (see Sect. 2.1) to select most relevant, descriptive terms from the ranked list of documents describing source code elements. That is, after the

[4] Several approaches, e.g., [4, 9], address the problem of input query definition. They consider not only the query but also related terms when evaluating the document models. As discussed earlier, these approaches are out of the scope of this chapter.

documents are ranked based on their similarity to the input query using LSI, as in [23, 24], the system selects the first n documents and ranks all terms that appear *uniquely* in these documents. The ranking is based on the similarity between each term and the document of the corpus, such that the terms that are similar to those in the selected n documents but not to the rest are ranked higher. Terms that are similar to documents not in the selected n results are penalized because they might be identifiers for data structures or utility classes which would pollute the top ranked list of terms. For the example in Fig. 1, given the LSI ranking with respect to the *automatic save file* query shown in Table 1, if n is set to 2, documents d_1 and d_2 are selected. The unique terms in these are "automatic," "do," and "run," all ranked high as they are not similar to any of the terms in the rest of the documents.

After the unique terms are ranked, the system selects the top k terms (attributes) from the first n documents (objects) and applies FCA (see Sect. 2.1) to build the set of concepts. For the three terms in our example, the concepts are $(\{d_1, d_2\}$, {`automatic, do`}), and $(\{d_1\}$, {`automatic, do, run`}). The terms describe the resulting documents. The user can inspect the generated concepts—the description and links to actual documents in the source code—and select those that are relevant. Similar to [23, 24], the technique requires a low amount of user interaction, as indicated by "+" in Fig. 8.

Shao et al. [36] introduce another approach that extends the work of Marcus et al. [23, 24] by completing the LSI ranking with static call graph analysis. Each method of the analyzed program is represented by a document containing its identifiers. After the LSI rank for each document with respect to the input query is calculated, the system builds a set of methods corresponding to documents ranked above a certain threshold and computes a set of all callers and callees of these methods. The LSI score of the elements in the computed set is augmented to represent their call graph proximity to one of the methods ranked high by LSI. The algorithm outputs a list of all methods organized in a descending order by their combined ranking. For the example in Fig. 1, element #3 is ranked low by LSI with respect to the query "*automatic save file*" (−0.2034 in Table 1). However, it is called by element #1 which has a high LSI rank (0.6319 in Table 1). Thus, the score of element #3 will be augmented and it will be ranked higher.

The technique requires no user interaction except defining and refining the input query describing the feature of interest, as indicated by "+" in Fig. 8.

Hill et al. [18] combine call graph traversal with the *tf-idf*-based ranking (see Sect. 2.3). The analyzed program is represented as a call graph and a set of text documents. Each document corresponds to a method of the program and includes all identifiers used in the method. The user provides an initial query that describes the feature, a seed method from which the exploration starts, and the exploration depth which determines the neighborhood to be explored (i.e., a maximal distance of explored methods from the seed).

Starting from the input seed method, the system traverses the program call graph and calculates the relevance score of each explored method by combining the following three parameters: (1) the *tf-idf* score of the identifiers in the method name; (2) the *tf-idf* score of the identifiers in the method body; and (3) a binary

parameter specifying whether the method is from a library or part of the user code. If the score of a method is higher than a preset relevance threshold, the method is marked as relevant. If the score is higher than a preset exploration threshold (which is usually lower than the relevance threshold) and the distance of the element from the seed is lower than the exploration depth, the system continues exploring the neighborhood of this element. Otherwise, the element becomes a "dead-end," and its neighborhood is not explored. When there are no additional elements to explore for the given exploration depth, the system outputs the call-graph neighborhood of the seed method in which all elements are scored and relevant elements are marked.

For the example in Fig. 1, if the element #1 is used as a seed and the exploration depth is set to 3, all elements other than #2 can be potentially explored. For the sake of the example, we disregard the terms that appear in method bodies and assume that the program does not use any binary methods. In such a case, the calculated score of element #3 is based on the *tf-idf* similarity of the method name to the input query—0.12 for the input query "*automatic save file*," as shown in Sect. 2.3. Thus, setting the exploration threshold above this value results in not exploring the part of the graph starting with element #1, and thus no elements are returned to the user. The exploration threshold of up to 0.12 results in further exploration of the call graph.

The relevance threshold specifies which of the explored elements are considered relevant. Both relevance and exploration thresholds are set empirically, based on the experience with programs under analysis. The technique requires no user interaction besides the definition and the refinement of the input feature description and seed method, and thus is marked with "+" in Fig. 8.

Chen et al. [7] present a technique for retrieving lines of code that are relevant to an input query by performing textual search on the cvs comments associated with these lines of code. The analyzed program is represented as a set of lines for a newest revision of each file. The system examines changes between subsequent versions of each file using the *cvs diff* command and associates the corresponding comment with each changed line. It stores all associated cvs comments for each line of a file in a database and retrieves all lines whose cvs comments contain at least one of the input query's words. The results are scored to indicate the quality of the match: the more query words appear in the comment, the higher is the score. In addition, the system searches the source code to find lines containing at least one of the query's words. It outputs a sorted list of files so that those with the highest number of matches appear first. Within each file, a sorted list of all lines that either match the query or are associated with a cvs comment that matches it is presented.

The technique is largely automated and requires no user interaction other than providing and possibly refining the input query, as indicated by "+" in Fig. 8.

4.2 Dynamic Feature Location Techniques

In this section, we describe techniques that rely on program execution for locating features in source code. The majority of such techniques address the feature

location task for sequentially executed programs, thus we focus the section on those techniques. We note that some of the described approaches have been extended, e.g., [1, 13], to handling distributed and multi-threaded systems as well.

4.2.1 Plain Output

Widle et al. [42] introduced one of the earliest feature location techniques taking a fully dynamic approach. The main idea is to compare execution traces obtained by exercising the feature of interest to those obtained when the feature of interest is inactive. Towards this end, the program is instrumented so that the components executed on a scenario/test case can be identified. The granularity of components, e.g., methods or lines of code, is defined by the user. The user specifies a set of test cases that invoke each feature. The system runs all input test cases and analyzes their execution traces, identifying *common* components—executed by *all* test cases. In addition, for each feature, it identifies (1) *potentially involved* components—executed by *at least one* test case of the feature; (2) *indispensably involved* components—executed by *all* test cases of the feature; and (3) *uniquely involved* components—executed by *at least one* test case of the feature and not executed by *any* test case of the other features. The system outputs sets of potentially involved, indispensably involved and uniquely involved components for each feature, as well as the set of all common components.

For the example in Fig. 1, the execution trace of the *automatic save file* feature can be compared to that of the *manual save file* feature. In this case, elements #3, #5, and #6 are considered *common*, since the *automatic save file* feature relies on the execution of *manual save file* and, thus, these methods are executed in both scenarios. Element #1 is considered *uniquely involved* as it is executed by the *automatic save file* feature only.

Since the user is required to define two sets of scenarios for each feature—those that exercise it and those that do not, the technique requires heavy user involvement and we assess it as "+ + +" in Fig. 8.

Wong et al. [43] present ideas similar to [42]. Its main contribution is in analyzing data flow dependencies in addition to the control flow (method calls) and in presenting a user-friendly graphical interface for visualizing features.

Eisenbarth et al. [14] attempts to address one of the most significant problems of dynamic approaches discussed above—the difficulty of defining execution scenarios that exercise exactly one feature. Their work relies on the assumption that execution scenarios can implement more than one feature and a feature can be implemented by more than one scenario. The work extends [42] with FCA (see Sect. 2.1) to obtain both computation units for a feature as well as the jointly and distinctly required computation units for a set of features.

The analyzed program is represented by an instrumented executable and a static program dependence graph whose nodes are methods, data fields, classes, etc. and whose edges are function calls, data access links, and other types of relationships obtained by static analysis. While in general the technique is applicable

to computation units on any level of granularity, the approach is implemented and evaluated for method-level components. The system first executes all given input scenarios, each of which can invoke multiple features. Optionally, users can identify special *start* and *end* scenarios whose components correspond to startup and shutdown operations and are excluded from all executions.

Users select a subset of execution scenarios they wish to investigate. Then, the approach uses FCA (see Sect. 2.1), where computation units are objects, scenarios are attributes and relationships specify whether a unit is executed when a particular scenario is performed, to create a concept lattice. Based on the lattice, the following information is derived: (1) a set of computation units *specific* to a feature—those used in all scenarios invoking the feature, but not in other scenarios; (2) a set of computation units *relevant* to a feature—used in all scenarios invoking the feature, and possibly in other scenarios; (3) a set of computation units *conditionally specific* to a feature—those used in some scenarios invoking the feature, but not in scenarios that do not invoke the feature; (4) a set of computation units *conditionally relevant* to a feature—those used in some scenarios invoking the feature, and possibly in other scenarios that do not invoke the feature; and (5) a set of computation units *irrelevant* to a feature—those used only in scenarios that do not invoke the feature. In addition, for each feature, the system builds a *starting set* in which the collected computation units are organized from more specific to less. It also builds a subset of the program dependency graph containing all transitive control flow successor and predecessors of computation units in the starting set (i.e., method callers and callees). The graph is annotated with features and scenarios for which the computation units were executed.

The user inspects the created program dependency graph and source code in the order suggested by the starting set, in order to refine the set of identified computation units for a feature by adding and removing computational units. During the inspection, the system also performs two further analyses to assist with the call graph inspection: *strongly connected component analysis* and *dominance analysis*. The former is used for identifying cycles in the dependency graph. If there is one computation unit in the cycle that contains feature-specific code, all computation units of the cycle are related to the feature because of the cycle dependency. The purpose of the latter is to identify computation units that must be executed in order to reach one of the computation units containing feature-specific code. All such computation units are related to the feature as well.

At the end of the process, a set of components deemed relevant for each feature is generated. Even though the technique attempts to assist the user in defining input scenarios, the required level of user interaction in defining the scenarios, selecting the order in which the scenarios are processed, as well as interactively inspecting and refining the produced result is still high, as indicated by "+ + +" in Fig. 8.

Koschke et al. [20] extend the work of Eisenbarth et al. [14] by considering statement-level rather than method-level computation units.

Asadi et al. [2] propose an approach which combines IR, dynamic-analysis, and search-based optimization techniques to locate *cohesive* and *decoupled* fragments of traces that correspond to features. The approach is based on the assumptions that

methods responsible for implementing a feature are likely to share some linguistic information and be called close to each other in an execution trace.

For an input set of scenarios that exercise the features of interest, the system collects execution traces and prunes methods invoked in most scenarios (e.g., those related to logging). In addition, it compresses traces to remove repetition of one or more method invocations and keeps one occurrence of each method. Next, it tokenizes each method's source code and comments, removing special characters, programming language keywords and terms belonging to a stop-word list for the English language (e.g., "the", "is", "at"). The remaining terms are tokenized separating identifiers using known coding styles. The terms belonging to each method are then ranked using the *tf-idf* metric (see Sect. 2.3) with respect to the rest of the corpus. For the example in Fig. 1, when considering only terms of the method names, the term `mind` appears in all documents and thus is ranked 0, while the term `controller` appears only in one document (that corresponds to element #8) and thus gets a higher rank—0.9. The obtained *term-by-document co-occurrence matrix* is transformed to vectors in the LSI space (see Sect. 2.2). A cosine similarity between two vectors in LSI space is used as a similarity measure between the corresponding documents (methods).

Next, the system uses *genetic optimization algorithm* [17]—an iterative procedure that searches for the best solution to a given problem by evaluating various possible alternatives using an *objective function*, in order to separate each execution trace into conceptually cohesive *segments* that correspond to the features being exercised in a trace. In this case, an optimal solution is defined by two objectives: maximizing *segment cohesion*—the average similarity between any pair of methods in a segment, and minimizing *segment coupling*—the average similarity between a segment and all other segments in a trace, calculated as average similarity between methods in the segment and those in different ones. That is, the algorithm favors merging of consecutive segments containing methods with high average similarity.

The approach does not rely on comparing traces that exercise the feature of interest to those that do not and does not assume that each trace corresponds to one feature. Thus, the task of defining the execution scenarios is relatively simple. However, the approach does not provide any assistance in helping the users to understand the meaning of the produced segments and tracing those to the features being exercised in the corresponding scenario; thus, this step requires a fair amount of user interaction. In Fig. 8, we rate this approach as "++".

4.2.2 Guided Output

Eisenberg et al. [15], similar to Eisenbarth et al. [14], present an attempt to deal with the complexity of scenario definition. The approach assumes that the user is unfamiliar with the system and thus should use pre-existing test suites, such as those typically available for systems developed with a test-driven development (TDD) strategy. It accepts as input a test suite that has some correlation between features and test cases (i.e., all features are exercised by at least one test case). Tests that exhibit some part of a feature functionality are mapped to that feature and referred

to as its *exhibiting test set*. Tests which are not part of any exhibiting test set are grouped into sets based on similarity between them and are referred to as the *non-exhibiting test set*.

For each feature, the system collects execution traces obtained by running all tests of the feature's exhibiting test set and generates a *calls set* which lists <*caller*, *callee*> pairs for each method call specified in the collected traces. It then ranks each method heuristically based on the following parameters: (1) *multiplicity*—a relationship between the percentage of tests in the exhibiting test set of the feature that execute the method and the percentage of tests in non-exhibiting test sets that execute that method; (2) *specialization*—the percentage of test sets that exercise the method. (If a method is exercised by many test sets, it is more likely to be a utility method); and (3) *depth*—the call depth (the number of stack frames from the top) of the method in the exhibiting test set compared to that in non-exhibiting test sets. The rationale behind these heuristics is that the exhibiting test set focuses on the feature in the most direct way. This is correlated with the call depth of the methods that implement this feature—the more "directly" a method is exercised, the lower its call depth.

For each feature, both the ranked list of methods and the generated *call set* are returned to the user. The goal of the former is to rank methods by their relevance to a feature, whereas the goal of the latter is to assist the user in understanding *why* a method is relevant to a feature. With respect to the required level of user interaction, we assess the technique as "++" in Fig. 8 because of the effort involved in creating test scenarios, if they are not available.

Poshyvanyk et al. [27] combine the techniques proposed in Marcus et al. [24] and Antoniol et al. [1] to use LSI (see Sect. 2.2) and execution-trace analysis to assist in feature location. The analyzed program is represented by a set of text documents describing software methods and a runnable program instrumented so that methods executed on any scenario can be identified.

Given a query that is formulated by the user to identify a given feature and two sets of scenarios—those that exercise the feature of interest and those that do not, the system first ranks input program methods using LSI. Then, it executes input scenarios, collects execution profiles and ranks each executed method based on the frequency of its appearance in the traces that exercise the feature of interest versus traces that do not. The final rank of each method is calculated as a weighted sum of the above two ranks. The system outputs a ranked list of methods for the input feature.

For the example in Fig. 1, element #1 is executed only in scenarios that exercise *automatic save file*. Thus, its LSI score (0.6319, as calculated in Table 1) will be increased, while the score of element #5 (0.2099, as calculated in Table 1) will be decreased to reflect the fact that it is executed in both scenarios that exercise the *automatic save file* feature and those that do not.

Similar to other dynamic approaches, this approach requires an extensive user involvement for defining scenarios that exercise the feature of interest and those that do not and, therefore, we assess the level of the necessary user interaction for this technique as "+ + +" in Fig. 8.

Liu et al. [22], similar to Poshyvanyk et al. [27], combine the use of LSI and execution-trace analysis. However, this work proposes operating on a *single* trace rather than on multiple traces that exercise/do not exercise the feature of interest.

Given a query that is formulated by the user to identify a feature of interest and a *single* scenario capturing that feature, the system executes the input scenario and ranks methods executed in the scenario using LSI with respect to the input query as in [24]. A ranked list of executed methods is returned to the user. For our example in Fig. 1, a scenario that executes the *automatic save file* feature invokes elements #1, #3, #6, and #7. These elements are returned to the user together with their LSI ranking, shown in Table 1.

Since the user is only required to provide a single scenario that exercises each feature of interest and a natural language description of that feature, we assess the level of the necessary user interaction for this technique as "+" in Fig. 8.

Rohatgi et al. [33] present a technique that is based on dynamic program analysis and static program dependence graph analysis. The technique operates on a class level, where the analyzed program is represented by an instrumented executable and a static program class dependency graph whose nodes are classes and whose edges are dependency relationships among these classes such as method calls, generalization, and realization.

As input, the system obtains a set of scenarios that invoke the features of interest. It executes all input scenarios, collects execution profiles on a class level, and uses *impact analysis* to score the relevance of the classes to the feature of interest: classes that impact many others in the system are ranked low as these classes are likely not feature-specific but rather "utility" classes implementing some core system functionality. The technique outputs a set of classes produced by the dynamic trace analysis, ranked by their relevance as calculated using impact analysis.

We assess the level of the necessary user interaction for this technique as "++" in Fig. 8 because it requires only a set of scenarios that invoke the features of interest and not those that don't.

Eaddy et al. [12] present the PDA technique called *prune dependency analysis* which is based on the assumption that an element is relevant to a feature if it should be *removed* or otherwise *altered* if the feature is removed from the program. The program is represented as a *program dependence graph* whose nodes are classes and methods, and whose edges are method invocations, containment relationships between a class and its methods, or inheritance relationships between classes. The system calculates the set of all elements affected by removing at least one element from the seed input set. For the example in Fig. 1, removing element #2 requires removing or altering element #4 that initiates a call to it in order to avoid compilation errors. Thus, element #4 is related to the feature that involves execution of element #2. Removing element #4 requires removing elements #7 and #8. The latter does not trigger any additional removals.

Furthermore, the work suggests combining the proposed technique with existing dynamic- and IR-based feature location approaches to achieve better accuracy. The dynamic feature location can use the approaches proposed in [15, 42] or others. These either produce a ranked set of methods, as in Eisenberg et al. [15] or an

unsorted list of relevant elements, as in Wilde et al. [42]. In the latter case, an element is assigned the score 1 if it is executed *only* by scenarios exercising the feature of interest, or 0 otherwise. The IR-based feature location uses the approach of Zhao et al. [44]: program elements are ranked with respect to feature descriptions (extracted from requirements) using the *vector space model*. It calculates the cosine of the distance between the corresponding vectors of terms, each of which first weighted using the *tf-idf* metric.

For each software element, the resulting score is calculated by normalizing, weighing and adding the similarity scores produced by the IR and the dynamic techniques, as in Poshyvanyk et al. [27]. Then, similar to Zhao et al. [44], the system applies a threshold to identify highly relevant elements. These are used as input to the *prune dependency analysis* which produces the set of additional relevant elements. The resulting set, ranked by the combination of scores produced by IR and dynamic techniques, is returned to the user.

For our example in Fig. 1, elements #1 and #2 are ranked high by the vector space model for the query "*automatic save file*." Since element #1 is executed only by scenarios that exercise the *automatic save file* feature, it is also ranked high by a dynamic analysis-based technique. Prune dependency analysis uses these two as the input seed set and adds elements #4, #7, and #8, so the result becomes {#1, #2, #4, #7, #8}. Since the technique requires two sets of scenarios for each feature—those that exercise it and those that do not, we assess the level of the necessary user interaction for this technique as "$+++$" (see Fig. 8).

Revelle et al. [29] propose improving the feature location accuracy by combining Similar to Liu et al. [22], the proposed system obtains as input a single scenario that exercises the feature of interest and a query that describes that feature. It runs the scenario and constructs a call graph from the execution trace, which is a subgraph of the static call graph and contains only the methods that were executed. Next, the system assigns each method of the graph a score using one of the existing web-mining algorithms—either HITS (see Sect. 2.4) or the PageRanked algorithm developed by Brin and Page [5], which is also based on similar ideas of citation analysis. The system then either filters out low-ranked methods (e.g., if the HITS authority score was used, as in Saul et al. [35]) or high-ranked methods (e.g., if the HITS hub score was used, as high-ranked methods represent common functions). The remaining set of elements is scored using LSI (see Sect. 2.2) based on their relevance to the input query describing the feature. The ranked list of these elements is returned to the user.

For the example in Fig. 1, elements #1, #3, #5, and #6 are invoked when the scenario exercising the *automatic save file* feature is executed. Assuming these elements are scored using HITS authority values, filtering out low-scored methods removes element #1 from the list of potentially relevant elements as its authority score is 0, as shown in Sect. 2.4. The remaining elements, #3, #5 and #6, are scored using LSI with respect to the query "*automatic save file*" (these scores are given in Table 1) and are returned to the user.

Similar to [22], since the user is only required to provide a single scenario for each feature of interest and a natural language description of that feature, we assess the level of the necessary user interaction for this technique as "+" in Fig. 8.

5 Which Technique to Prefer?

As the survey shows, there is large variety in existing approaches and implementation strategies for feature location. We believe that trying to identify a single technique that is superior to the rest would be impractical. Clearly, there is no "silver bullet," and the performance of each technique largely depends on its applicability to the analyzed input programs and the quality of the feature description (*feature intension*) provided by the user. In this section, we discuss considerations and provide explicit guidelines for practitioners who need to choose a particular feature location technique to apply.

The chosen technique should first and foremost be suitable to the program being analyzed: specifically, if the studied program contains no documentation and no meaningful identifier names, IR-based feature location techniques will be unable to achieve high-quality results. Similarly, if the implementation of a feature is spread across several program modules or is hooked into numerous extension points provided by the platform on which the program is built (e.g., invoking methods via event queues), techniques based on program dependency analysis will either be unable to find all elements that relate to the implementation of the feature or will find too many unrelated elements. When program execution scenarios are unavailable or it is cumbersome to produce scenarios that execute a specific set of features (e.g., because the feature of interest is not a functional feature that is "visible" at the user level), dynamic feature location techniques will not be applicable. Figure 9 assesses the surveyed feature location techniques based on the above selection criteria.

For our example in Figs. 1 and 2, program elements have meaningful names ("file" vs. "f" or "property" vs. "prp"). Thus, it is reasonable to choose one of the techniques that rely on that quality, as marked in the corresponding column of Fig. 9. Since the implementation of the Freemind software is asynchronous and relies on event queues to perform method invocation, techniques that analyze call graph dependency might be less efficient. In addition, defining a scenario that triggers the *automatic save file* feature might not be trivial—there is no user operation that directly invokes the automatic save (as opposed to the manual save) functionality. Therefore, techniques that do not require program execution are a better choice which leads us to the approaches in Shepherd et al. [38], Marcus et al. [23, 24], or Poshyvanyk et al. [26].

With respect to the quality of a feature intent provided by the user, IR-based techniques are usually most sensitive to the quality of their input—the query that describes the feature of interest. The results produced by these techniques are often as good as the query that they use. Input query definition and the user assistance during that process are further discussed by [4, 9] and others. Techniques based on

		Technique	Strongly Coupled Implementation	Meaninful Names	Change Histories	Execution Scenarios
Static	Plain	**Chen et al. [8]**	√			
		Walkinshaw at al. [41]	√			
		Shepherd et al. [38] (Find-concept)		√		
		Zhao et al. [44] (SNIAFL)	(√)	√		
		Robillard et al. [31]			√	
		Trifu [40]	(√)			
	Guided	**Robillard [30]** (Suade)	√			
		Saul et al. [35]	√			
		Marcus et al. [23, 24]		√		
		Poshyvanyk et al. [26]		√		
		Shao et al. [36]	(√)	√		
		Hill et al. [18] (Dora)	√	√		
		Chen et al. [7]			√	
Dynamic	Plain	**Wilde et al. [42]** (Sw. Reconnaissance)				√
		Wong et al. [43]				√
		Eisenbarth et al. [14]	(√)			√
		Koschke et al. [20]	(√)			√
		Asadi et al. [2]		√		√
	Guided	**Eisenberg et al. [15]**				√
		Poshyvanyk et al. [27]		√		√
		Liu et al. [22] (SITIR)		√		√
		Rohatgi et al. [33]				√
		Eaddy et al. [12] (Cerberus)	√	√		√
		Revelle et al. [29]		√		√

Fig. 9 Criteria for selecting a feature location technique

comparing dynamic execution traces are also sensitive to the nature of their input—if execution scenarios do not cover all aspects of the located feature, the accuracy of the feature location will likely be low.

The approaches also differ in the required level of user interaction (see the last column of Fig. 8). We assess the level of user interaction based on the *effort* that the user has to invest in *operating* the technique. This includes the effort involved in defining the input feature intension (e.g., a set of scenarios exercising the features of interest), interactively following the location process (e.g., filtering intermediate results produced by the technique) and interpreting the produced results (e.g., mapping retrieved variables to the code statements that use them).

Since more highly automated techniques are easier to execute, their "barrier to entry"—the effort required to produce the initial approximation of the result—is lower and thus their adoption is easier. On the other hand, the techniques that require more user interaction are usually able to produce better results because they harvest this "human intelligence" for the feature location process.

Furthermore, automated techniques could be a better choice for the users that seek an "initial approximation" of the results and are able to complete them manually since they are familiar with the analyzed code. On the other hand, users that cannot rely on their understanding of the analyzed code should probably choose a technique that is more effective at producing relevant results, even though operating such a technique requires a more intensive investment of time and effort.

6 Summary and Conclusions

In this chapter, we provided a detailed description of 24 feature location techniques and discussed their properties. While all of the surveyed approaches share the same goal—establishing traceability between a specific feature of interest that is specified by the user and the artifacts that implement that feature, their underlying design principles, their input, and the quality of the results which they produce differ substantially. We discussed those in detail and identified criteria that can be used when choosing a particular feature location technique in a practical setting. We also illustrated the techniques on a common example in order to improve the understandability of their underlying principles and implementation decisions.

Even though the area of feature location is mature, there is variety in existing techniques, which is caused by the common desire to achieve high accuracy: automatically find a high number of relevant elements (high *recall*) while maintaining a low number of false-positive results (high *precision*). As discussed in Sect. 5, since there is no optimal technique, each of the approaches proposes heuristics that are applicable in a particular context, making the technique efficient in these settings.

Feature Location for SPLE. In the context of product line engineering, identifying traceability between product line features and product artifacts that realize those features is an essential step towards capturing, maintaining, and evolving well-formed product line systems. Traceability reconstruction is also an important step when identifying product line architectures in existing implementations.

Each of the existing feature location techniques can be used for detecting features of products that belong to a product family. Feature location is done while treating these products as singular independent entities. Yet, considering families of *related* products can provide additional input to the feature location process and thus improve the accuracy of the techniques by considering product line commonalities and variations.

When considering a specific feature that exists only in some products of the family, comparing the code of a product that contains the feature to the code of the one that does not can partition the code into two parts: *unique* to the product and *shared*. This partitioning can help detect relevant elements with higher accuracy because it limits the results to the elements of the *unique* part where the feature of interest is located. For example, it can be used to filter out irrelevant elements (those that belong to the shared parts of the code) from the program execution trace analyzed by Liu et al. [22].

The above partitioning can also improve scoring and traversal mechanisms employed by existing feature location techniques when *searching* for these relevant elements. For example, it can be used for augmenting the score calculation formula used by Hill et al. [18] so that the score of elements belonging to the shared parts of the code is decreased while the score of those in the unique parts is increased, as shown in [34]. This affects the call graph traversal process and the ability of the algorithm to reach the desired elements, while avoiding passes that lead to

false-positive results. More complex partitioning, obtained by comparing multiple products to each other, can provide even better solutions.

In addition, it might be interesting to develop methods for incremental analysis of product lines, where the traceability links obtained for one variant may be carried over to the next variant. This will prevent unnecessary re-analysis and leverage the effort and human intelligence invested in one product for more efficient feature location in others. We explore this and other directions in our ongoing work.

References

1. Antoniol, G., Guéhéneuc, Y.G.: Feature identification: an epidemiological metaphor. IEEE TSE **32**, 627–641 (2006)
2. Asadi, F., Di Penta, M., Antoniol, G., Guéhéneuc, Y.G.: A heuristic-based approach to identify concepts in execution traces. In: Proc. of CSMR'10, pp. 31–40, 2010
3. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman, Boston (1999)
4. Bai, J., Song, D., Bruza, P., Nie, J.Y., Cao, G.: Query expansion using term relationships in language models for information retrieval. In: Proc. of CIKM'05, pp. 688–695, 2005
5. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proc. of WWW7, pp. 107–117, 1998
6. Brooks, F.P. Jr.: No silver bullet essence and accidents of software engineering. IEEE Comput. **20**, 10–19 (1987)
7. Chen, A., Chou, E., Wong, J., Yao, A.Y., Zhang, Q., Zhang, S., Michail, A.: CVSSearch: Searching through source code using CVS comments. In: Proc. of ICSM'01, 2001
8. Chen, K., Rajlich, V.: Case study of feature location using dependence graph. In: Proc. of IWPC'00, pp. 241–249, 2000
9. Cleary, B., Exton, C., Buckley, J., English, M.: An empirical analysis of information retrieval based concept location techniques in software comprehension. J. Empir. Software Eng. **14**, 93–130 (2009)
10. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley Longman, Boston (2001)
11. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. Evol. Process **25**(1), 53–95 (2013)
12. Eaddy, M., Aho, A.V., Antoniol, G., Guéhéneuc, Y.G.: CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: Proc. of ICPC'08, pp. 53–62, 2008
13. Edwards, D., Simmons, S., Wilde, N.: An approach to feature location in distributed systems. J. Syst. Software **79**, 57–68 (2006)
14. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE TSE **29**, 210–224 (2003)
15. Eisenberg, A.D., De Volder, K.: Dynamic feature traces: finding features in unfamiliar code. In: Proc.of ICSM'05, pp. 337–346, 2005
16. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, New York (1999)
17. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman, Boston (1989)
18. Hill, E., Pollock, L., Vijay-Shanker, K.: Exploring the neighborhood with dora to expedite software maintenance. In: Proc. of ASE'07, pp. 14–23, 2007

19. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. J. ACM **46**, 604–632 (1999)
20. Koschke, R., Quante, J.: On dynamic feature location. In: Proc. of ASE'05, 2005
21. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Process. **25**(2–3), 259–284 (1998)
22. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: Proc. of ASE'07, 2007
23. Marcus, A.: Semantic-driven program analysis. In: Proc. of ICSM'04, pp. 469–473, 2004
24. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An information retrieval approach to concept location in source code. In: Proc. of WCRE'04, pp. 214–223, 2004
25. Pohl, K., Boeckle, G., van der Linden, F.: Software Product Line Engineering : Foundations, Principles, and Techniques. Springer, New York (2005)
26. Poshyvanyk, D., Marcus, A.: Combining formal concept analysis with information retrieval for concept location in source code. In: Proc. of ICPC'07, pp. 37–48, 2007
27. Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE TSE **33**, 420–432 (2007)
28. Qin, T., Zhang, L., Zhou, Z., Hao, D., Sun, J.: Discovering use cases from source code using the branch-reserving call graph. In: Proc. of APSEC'03, pp. 60–67, 2003
29. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: Proc. of ICPC'10, pp. 14–23, 2010
30. Robillard, M.P.: Automatic generation of suggestions for program investigation. In: Proc. of ESEC/FSE-13, pp. 11–20, 2005
31. Robillard, M.P., Dagenais, B.: Retrieving task-related clusters from change history. In: Proc. of WCRE'08, pp. 17–26, 2008
32. Robillard, M.P., Shepherd, D., Hill, E., Vijay-Shanker, K., Pollock, L.: An empirical study of the concept assignment problem. Tech. Rep. SOCS -TR-2007.3, School of Computer Science, McGill University (2007)
33. Rohatgi, A., Hamou-Lhadj, A., Rilling, J.: An approach for mapping features to code based on static and dynamic analysis. In: Proc. of ICPC'08, pp. 236–241, 2008
34. Rubin, J., Chechik, M.: Locating distinguishing features using diff sets. In: Proc. of ASE'12, pp. 242–245, 2012
35. Saul, Z.M., Filkov, V., Devanbu, P., Bird, C.: Recommending random walks. In: Proc. of FSE'07, pp. 15–24, 2007
36. Shao, P., Smith, R.K.: Feature location by IR modules and call graph. In: Proc. of ACM-SE 47, pp. 70:1–70:4, 2009
37. Shepherd, D., Pollock, L., Vijay-Shanker, K.: Towards supporting on-demand virtual remodularization using program graphs. In: Proc. of AOSD'06, pp. 3–14, 2006
38. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: Proc. of AOSD'07, pp. 212–224, 2007
39. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. **3**(3), 121–189 (1995)
40. Trifu, M.: Improving the dataflow-based concern identification approach. In: Proc. of CSMR'09, pp. 109–118, 2009
41. Walkinshaw, N., Roper, M., Wood, M.: Feature location and extraction using landmarks and barriers. In: Proc. of ICSM'07, pp. 54–63, 2007
42. Wilde, N., Scully, M.C.: Software reconnaissance: mapping program features to code. J. Software Mainten. **7**, 49–62 (1995)
43. Wong, W.E., Horgan, J.R., Gokhale, S.S., Trivedi, K.S.: Locating program features using execution slices. In: Proc. of ASSET'99, pp. 194–203, 1999
44. Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F.: SNIAFL: towards a static noninteractive approach to feature location. ACM TOSEM **15**, 195–226 (2006)

Modeling Real-Time Design Patterns with the UML-RTDP Profile

Saoussen Rekhis, Nadia Bouassida, Rafik Bouaziz, Claude Duvallet, and Bruno Sadeg

Abstract The use of design patterns in the real-time (RT) domain could bring many benefits. RT design patterns capture domain knowledge and design expertise. They improve the quality of RT software. These patterns contain a set of common elements and a set of variable elements known as variation points. Yet one of the main difficulties for representing RT design patterns is variability management. Thus, many questions arise: how to express variations? How to ensure the consistency of various views and avoid conflicts?

In order to express the variability in an RT design pattern and to reinforce its comprehension, it is necessary to define a design language that aims to model RT features and to distinguish the commonalities and differences between different RT applications involving time-constrained data and time-constrained transactions. Accordingly, we present in this chapter new UML extensions that take into account the design of both RT-specific concepts and variability in patterns. The coherence between the proposed extensions is then ensured by OCL (object constraint language) constraints. Finally, the UML extensions are illustrated using an example of a controller pattern.

Keywords Design patterns • Real-time applications • UML extensions

S. Rekhis (✉) • N. Bouassida • R. Bouaziz
MIRACL-ISIMS, Sfax University, BP 1088, 3018, Sfax, Tunisia
e-mail: saoussen.rekhis@fsegs.rnu.tn; nadia.bouassida@isimsf.rnu.tn; raf.bouaziz@fsegs.rnu.tn

C. Duvallet • B. Sadeg
LITIS, UFR des Sciences et Techniques, BP 540, 76 058, Le Havre Cedex, France
e-mail: claude.duvallet@univ-lehavre.fr; bruno.sadeg@univ-lehavre.fr

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_3, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

Design patterns [1] are solutions to common problems in software design. Their use can help software designers obtain software with enhanced quality. Design patterns that are specific for a particular domain are called domain-specific patterns [2]. They factorize parts of models, common to all systems in the same domain, and express the differences through variable elements.

Real-time (RT) design patterns constitute an example of domain-specific patterns that encapsulate the RT domain requirements. In fact, it is necessary to give prominence to the modeling of RT applications which have to meet RT constraints, i.e. they have to guarantee that each action (transaction) meets its deadline and that data are used during their validity interval. We need to express RT design patterns to reduce the complexity of RT application design and to provide a common vocabulary for computer scientists across the RT domain.

In order to benefit from RT design pattern advantages, it is necessary to have an expressive design language that establishes a relation between the different systems needing similar facilities and that provides a good ground to build on in order to create a design language for RT patterns. The design language should be complete enough to deal with RT design pattern representation at the specification and the instantiation levels.

At the specification level, the design language must (1) cope with pattern variability, (2) ensure variation consistency in static and dynamic models, and (3) take into account requirements and specificities of the RT domain itself. In fact, RT domain has many details that must be taken into account by the design pattern notation since it is crucial to reflect the current state of the controlled environment and to meet constraints of RT data and RT transactions. For instance, in a freeway traffic management application reusing design patterns, it is essential to distinguish passive resources from RT active resources. Thus, the pattern design language must specify RT features specific to this type of applications. It must, also, show passive components (like vehicle speed, traffic volume, and segment occupancy) which need to be set and controlled by active resources. The vehicle speed passive component provides an RT service called updateSpeed. This operation carries the concurrency kind (*writer*) and the execution kind (*remoteImmediate*) indicating that the execution is performed immediately with the called active object (Controller). Besides, the active controller resource creates dynamically schedulable resources to handle the execution of its services needing to be achieved before a deadline. To summarize a design language expressing RT patterns must express the specificities of the RT domain, for example, it must provide extensions differentiating between passive resources and RT active resources. It must, also, specify RT features such as the concurrency kind, the relative deadline, and the acceptable deadline miss ratio.

At the instantiation level, the design language has to clearly identify the elements belonging to each design pattern and to show the role played by each pattern element in order to avoid ambiguity when composing patterns.

This chapter proposes a new UML-profile, named UML-RTDP that extends UML with concepts related to RT design patterns. It integrates, also, OCL constraints ensuring variation points consistency. The motivations behind the proposed extensions are threefold. The first motivation is to have flexible patterns that distinguish the fixed elements from variable elements in the pattern. The second motivation is to facilitate the comprehension of design patterns. The third motivation is to model RT applications constraints and their nonfunctional properties.

The remainder of this chapter is organized as follows. Section 2 overviews and evaluates currently proposed design languages and their extensions. Section 3 presents our UML profile that models RT design patterns. Section 4 defines a set of well-formedness rules written in OCL (object constraint language) [3], in order to verify the RT design patterns correctness and consistency. Section 5 illustrates the design language with an RT controller pattern and presents an example of a freeway traffic management system reusing it. Section 6 describes the implementation of the proposed profile. Section 7 concludes the chapter and outlines future work.

2 Overview of Current Work

This section provides an overview of current design languages for pattern representation. We define a set of criteria necessary for pattern notations and then we present their advantages and limits. Second, we briefly present in Sect. 2.2 the RT profiles and the UML extensions taking into account the real-time system requirements.

2.1 Overview of UML Extensions for Design Patterns Representation

Design patterns are proposed in order to be reused. Therefore, their design must be generic in order to be instantiated. Moreover, their design must show variations, which will be adopted for a certain type of application and not adopted for others. Thus, the expression of variability when representing patterns is very important. In addition, patterns are proposed by pattern designers at the specification level and they are reused by application designers at the instantiation level. This fact implies that the expressivity of their design is essential. Finally, any pattern design consists in a static and dynamic view and as a consequence the consistency and coherence between these views is crucial to ensure a good comprehension of the pattern and thus its correct reuse.

For all these reasons, two categories of criteria have to be taken into account to evaluate the currently proposed languages for pattern representations: criteria for the specification of design patterns and others for their instantiation.

2.1.1 Criteria Definition for Design Pattern Representation

– Criteria for design pattern representation at the specification level

  C1. Expressivity: Design patterns have mostly been described using natural language, complex mathematical, or logic-based formalisms [4, 5] which are not easily understood by an inexperienced designer. This leads to complications in incorporating design patterns efficiently into the modeling of a new system. To remediate to this difficulty, the solution is using an expressive visual notation based on UML to specify patterns. In addition, it is essential to differentiate the notations used for pattern representation at specification level from those used for pattern instantiation in order to enhance the expressivity of design language.

  This criterion highlights the usefulness of UML to improve the pattern specification quality because UML allows ease in visualizing, defining, and documenting the artifacts of the system under development.

  C2. Variability: Variability in a model is the representation of items (e.g., classes, attributes, and associations) that can vary according to a specific context. The design language has to express variability in order to guide the designer in determining the variable elements that may differ when applying the patterns. In fact, variability is classified into optional and alternative characteristics. So, it is important to show the optional elements which can be omitted in a pattern instance. It is also necessary to clarify the elements that can vary according to a specific context.

  This criterion implies that the design language has to show clearly the variable items in a pattern, in order to help the designer in choosing the adapted variation in a pattern instantiation and to reduce the probability of pattern misuse.

  C3. Consistency: In accordance with the separation of concerns principle, a design pattern is often described via several complementary views: class diagrams, sequence diagrams, etc. Such views are dependent on each other. In fact, when a pattern element becomes variable in one view, these variations may have an impact on the other views. For example, a fundamental class in a class diagram (i.e., a class representing the essence of the pattern and that must belong to any application reusing the pattern) must have a corresponding fundamental object in the sequence diagram. The correct specification of patterns depends on respecting the rules inherent to the variability consistency management. These rules are specified by constraints that are generally expressed in OCL.

  In our case, the consistency criterion implies the definition of constraints that maintain coherence between the structural view (class diagram) and the behavioral view (sequence diagram).

– Criteria for design pattern representation at the instantiation level

  C1. Traceability: The traceability criterion consists of ease in identifying design patterns when they are applied and integrated with other patterns. In fact, we not only need to identify each pattern in a design, but also we want to show the methods and attributes that play important roles in the pattern. Explicit

representation of the key methods and attributes can assist on the traceability of a pattern since it allows us to trace back to the design pattern from a complex design diagram [6].

C2. Composition: A design pattern may be composed or integrated with other patterns to solve multiple design problems in a software application. Thus, when several patterns are combined in a design, the pattern design language must clearly distinguish among the elements belonging to each design pattern. This is particularly important, when there are overlapping parts between the composed patterns.

Note that the development of applications using design patterns requires a careful look at composition techniques, which are categorized as: behavioral composition techniques and structural composition techniques. Indeed, the behavioral techniques show how dynamic specifications of patterns can be composed using sequence diagram, whereas structural techniques show how the static architectural specifications of instantiated patterns can be composed using a class diagram [7].

2.1.2 Comparison of UML Notations

There are several UML notations that proposed extensions to present general design patterns and domain models. Many of them can be used to express concepts relative to domain-specific design patterns such as their flexibility. In the following, we present a comparison of the most recent notations, using the specification and instantiation criteria.

The UML profile proposed by Dong et al. [8] focuses more on the pattern applicability context than on the pattern specification. It proposes new stereotypes and tagged values for the explicit representation of design patterns in software designs. These extensions display the pattern name, the role names of the classes, the attributes and the operations in the pattern and show how many instances of a design pattern are composed. That is, when two or more classes represent the overlapping part of the composition, the proposed notation shows the roles that these classes play in each pattern. However, the proposed profile does not deal with the behavioral view: it does not provide support on how to keep track of the interactions when a pattern is instantiated. In addition, it does not express the variability of patterns and does not specify the constraints delimiting the pattern applicability.

Figure 1 shows an example instantiating the composite design pattern [1] represented with the profile of Dong et al. [8]. The roles that the classes and operations play in the pattern are described, respectively, with <<patternClass>> and <<patternOperation>> stereotypes. Thus, this profile satisfies the traceability and composition criteria but it fails to satisfy the criteria necessary to the pattern specification level.

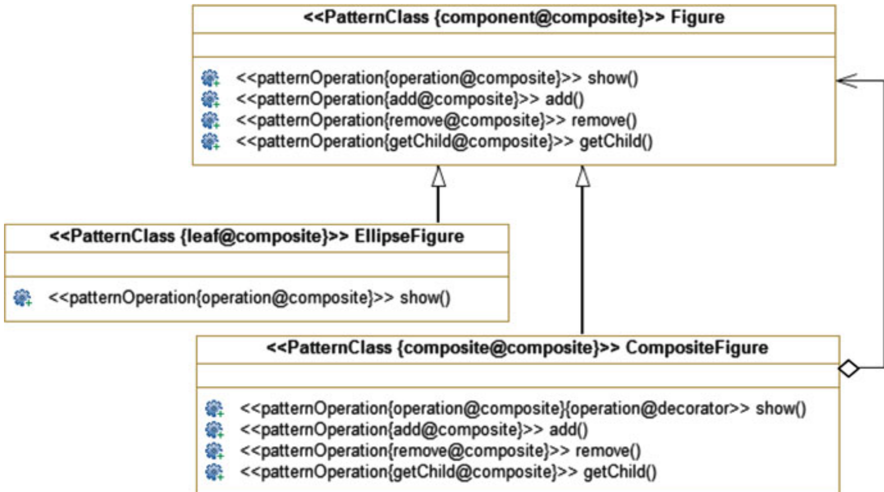


Fig. 1 Example showing the instantiation of a composite design pattern [8]

Unlike the previous work, P-UML profile [6] proposes extensions showing the pattern variation points only in a class diagram and guiding the designer in instantiating a pattern. It defines two tagged values to express pattern variability:

– {*variable*} Tagged value indicates that the method implementation varies according to the pattern instantiation.
– {*extensible*} Tagged value indicates that the class interface may be extended by adding new attributes and/or methods.

The P-UML profile indicates also that new classes may be added during the pattern instantiation through the {*incomplete*} constraint applied on the generalization relationship. It provides support for traceability of pattern instantiation by using an ellipse in the bottom of a class that indicates the pattern name and the role through which this class participates in the pattern. However, the class diagram may seem to be overloaded since the proposed notation presents an association between ellipses to join the elements of the same pattern. To summarize, the P-UML profile does not distinguish between the extensions used in a pattern instantiation from those used in a pattern specification, this reduces the expressivity of the profile. In addition, the variability criterion is partially expressed since P-UML does not propose extensions to differentiate between the optional elements and the fundamental elements in a pattern. Finally, the consistency criterion is not verified since this profile focuses only on a structural view and does not define constraints to manage the impact of variable elements in a behavioral view.

Figure 2 represents a combination of the composite and the observer patterns as proposed by P-UML. As shown in this example, the class *Figure* is the overlapping element between the composed patterns. It plays the role of a subject in the *Observer*

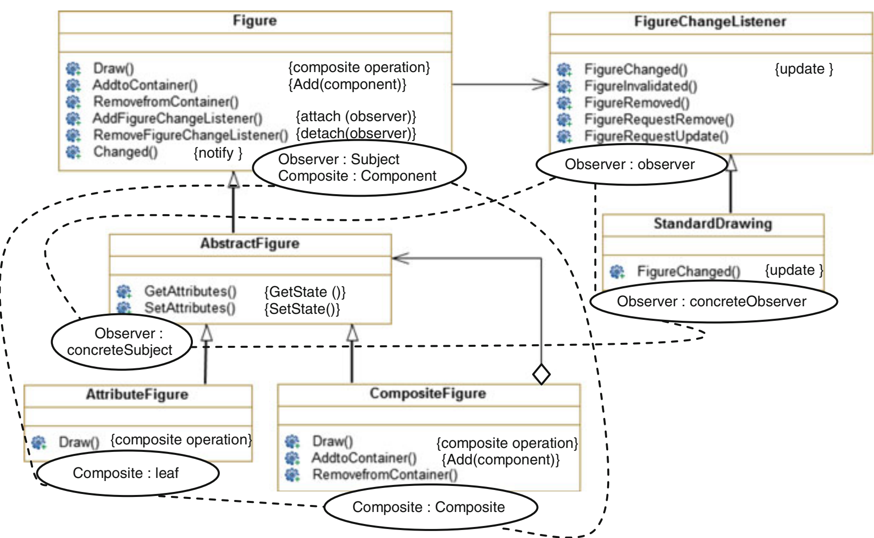


Fig. 2 Example of combination of composite and observer patterns [6]

pattern and it plays the role of a component in the *Composite* pattern. Thus, the traceability and composition criteria are verified by the P-UML profile.

Unlike all previous notations, the profile proposed by Arnaud et al. [9] focuses on the variability expression in the functional, dynamic, and static views. The use case diagram is the input model for the instantiation process, where the application designer selects functionality variants. However, the use case diagram is too abstract and cannot allow the designer to identify, for example, the optional attributes or methods according to its needs. Therefore, this profile partially fulfills the variability criterion, while it fails in covering the traceability and composition criteria. It does not permit the visualization and preservation of pattern-related information in a design model created through patterns instantiation. It also does not present mechanisms to compose either static or dynamic specifications of patterns. Besides, this profile is not very expressive since it proposes representation of the static view of a pattern with very elementary separated packages. Each package refers to a functionality variant.

While it is necessary to describe how the different roles of a pattern interact, the above presented notations lack a way to capture the behavioral information of design patterns. In order to fill these gaps, Loo and Lee [10] proposed an extension of the UML sequence diagram to allow designers to define and visualize the pattern roles and the different types of interaction groups for a design pattern. The authors proposed four stereotypes which are named <<PatternRole>>, <<PatternEngage>>, <<PatternDisengage>>, and <<PatternInteractionFragment>>. The first one is used to define the pattern role of a *Message* and *Lifeline* in a particular design pattern via the tag definition *role*. The three other stereotypes are used to specify,

Table 1 UML profiles comparison

		Dong et al., [8]	Bouassida et al., [6]	Arnaud et al. [9]	Loo et al., [10]	Reinhartz Berger et al. [11]
specification criteria	Expressivity	not verified	partially verified	partially verified	partially verified	well verified
	Variability	not expressed	partially expressed	partially expressed	partially expressed	well expressed
	Consistency	not verified	not verified	not verified	not verified	partially verified
instantiation criteria	Traceability	verified only for class diagrams	verified only for class diagram	not verified	verified only for sequence diagram	verified
	Composition	only the composition of patterns structural view is considered	only the composition of patterns structural view is considered	not verified	only the composition of patterns behavioral view is considered	not verified

respectively, that a new pattern role is added to a particular design pattern, a pattern role is removed from a particular design pattern, or a pattern role exists in a particular design pattern. Nevertheless, the proposed profile does not distinguish between the extensions used in pattern instantiation from those used in pattern specification, which reduces the expressivity of notations. Moreover, the variability is partially expressed within this profile since it does not propose extensions to represent variation points in the static view.

Similar to the proposed notations for design pattern representation, Reinhartz-Berger et al. [11] have proposed an Application-based DOmain Modeling approach (ADOM-UML) which provides new stereotypes to denote the multiplicity variability of the different domain model elements. The multiplicity stereotypes aim to represent how many times a model element can appear in a specific context. Particularly, the authors define four stereotypes: <<optional single>>, <<optional many>>, <<mandatory single>>, and <<mandatory many>>. Each stereotype has two associated tagged values, min and max, which define the lowest and the uppermost multiplicity boundaries. These stereotypes can be used for expressing variability in domain-specific design patterns.

Table 1 evaluates the different UML profiles with regard to the design pattern specification and instantiation criteria.

The comparison study recapitulated in Table 1 shows that none of the proposed UML profiles satisfies all the specification and instantiation criteria. Moreover,

none of them proposes OCL rules to ensure variation point consistency. Therefore, the validity of pattern models cannot be verified. The ADOM-UML approach is the only one that enables the designer to verify that the domain constraints are satisfied to validate the application model. However, it does not define constraints to deal with the dependence of variable elements in a domain model.

Besides, the studied works mainly focus on the pattern structure modeling and have limits in expressing variability and in representing pattern participant roles, essentially in the behavioral view. Note that the unique profile interested in behavior was proposed by Loo et al. [10]. It defines stereotypes to retrieve pattern behavioral information and to represent pattern variants as interaction alternatives. In addition, the studied profiles are not suitable for the RT design patterns representation since they do not provide extensions to fulfill the RT applications requirements, such as those necessary for the freeway traffic management example.

In summary, we consider that the definition of a UML profile focusing on RT design patterns representation and taking into account the, already, presented criteria as well as the specificities of the RT domain is necessary. This profile will allow us to have expressive, understandable, and consistent patterns.

2.2 Overview of UML Extensions for RT Applications

Several works have proposed UML extensions to take into account the real-time system requirements such as *RT-UML* [12] and *ACCORD/UML* [13]. The basic concepts of *RT-UML* were integrated in the UML standard through the UML profile for *S*chedulability, *P*erformance, and *T*ime (denoted SPT profile) [14]. Recently, *MARTE* profile [15] for *M*odeling and *A*nalysis of *R*eal-*T*ime *E*mbedded systems has been standardized by the OMG. It is intended to replace the existing UML Profile for SPT profile. MARTE defines extensions that provide high-level modeling concepts to deal with RT and embedded features modeling as well as specific modeling artifacts to be able to support both software and hardware execution.

Another work proposed the *UML-RTDB* profile [16] to express real-time database features in a structural model. Unlike the previous profiles, it supplies concepts for real-time database modeling such as RT attributes, RT methods, and RT classes. In addition, UML-RTDB specifies two kinds of real-time attributes, *sensor* attributes and *derived* attributes, in order to satisfy the requirements of current real-time applications. But some proposed stereotypes overlap with the UML extensions presented by MARTE profile especially those relative to the RT methods. In fact, the UML-RTDB stereotypes <<*Periodic*>>, <<*Sporadic*>>, and <<*Aperiodic*>> that express, respectively, periodic, sporadic, and aperiodic methods in the class diagrams, have the same meaning as the tagged value *Occurrence Kind* of the <<*rtFeature*>> stereotype defined in MARTE. Therefore, we adapt some MARTE stereotypes modeling RT aspects instead of the other UML extensions proposed for the modeling of RT applications since MARTE is a standardized profile.

Nevertheless, the use of UML extensions proposed for modeling RT application characteristics remains insufficient to model RT design patterns. That is, RT patterns must be generic designs intended to be specialized and reused by any application in the RT domain. For this reason, in addition to the UML extensions representing RT aspects, we need new notations distinguishing the commonalities and differences between applications in the pattern domain. Moreover, we need new concepts for the explicit representation of the pattern elements' roles for the purpose of traceability.

In the next section, we describe the extensions that we propose to take into account these new concepts.

3 UML Profile for RT Design Patterns

A design language specific for the representation of RT patterns has to support not only the flexibility characteristic of patterns but also the specificities of the RT domain itself. However, the UML standard only deals with the design of specific applications and it does take into account the modeling of RT application features in general and variations in particular. Therefore, different extensions to the original language have been proposed as presented in the previous section.

In this section, we propose a new notation for patterns, which is an extension of the "UML 2.1.2" [17]. The extensions put emphasis on the variability in a pattern. In addition, they outline the roles played by each pattern element in the application instantiating it and therefore allow the visual distinction between patterns. Moreover, the extensions allow the easy specification of RT applications constraints and their nonfunctional properties.

3.1 UML Extensions for Modeling RT Design Patterns

This subsection summarizes stereotypes showing the optional and fundamental elements participating in a pattern and assisting the designer in pattern reuse. We describe below the purpose of each stereotype:

- <<***optional***>> stereotype: It is inspired from <<optional single>> and <<optional many>> stereotypes defined in [11]. In fact, the variety of applications within the RT domain is quite large. For this reason, we cannot specify **exactly** how many times a pattern element can appear in a specific RT application. Thus, we use <<optional>> stereotype to represent the optional features (i.e., attribute or method) that can be omitted in a pattern instance.

  Each method or attribute which is not stereotyped <<optional>> in a fundamental classifier (i.e., class, interface, etc.) means that it is an essential element that plays an important role in the pattern.

- <<***mandatory***>> stereotype: It is inspired from <<mandatory single>> and <<mandatory many>> defined in [11]. We propose the <<mandatory>> stereotype to specify a fundamental element (class, association, aggregation, etc.) that must be instantiated at least once by the designer when he models a specific application. Besides, each pattern element which is not stereotyped <<mandatory>> means that it is an optional one, except the generalization relation that permits representation of alternative elements. All the attributes and methods of an optional class are implicitly optional.

  In the sequence diagram, the <<mandatory>> stereotype is applied to the *Lifeline* metaclass. It is used to model a fundamental object which is an instance of a classifier stereotyped <<mandatory>> in the class diagram.
- <<***extensible***>> stereotype: It is inspired by {extensible} tagged value proposed in [6]. It indicates that the class interface may be extended by adding new attributes and/or methods. Moreover, two properties related to the extensible stereotype are proposed, in order to specify the type of features (attribute or method) that may be added by the designer.
  - *extensibleAttribute* tag: It takes the value false to indicate that the designer cannot add new attributes when he instantiates the pattern. Otherwise, this tag takes the value true.
  - *extensibleMethod* tag: It indicates if the designer may add new methods when he instantiates the pattern. The default value is true.
- <<***variable***>> stereotype: It has the same meaning with the {variable} tagged value proposed in [6]. It indicates that the method implementation varies according to the pattern instantiation.

3.2 UML Extensions for Instantiating RT Design Patterns

Some of the existing notations (Dong & Yang UML profile [8] and P-UML profile [6]) provide support on how to keep track of the pattern when instantiated. These notations focus on the composition of generic design patterns (like GoF patterns [1]) which are intended to be instantiated in many contexts. Each instantiation may change the names of pattern classes, operations, and attributes according to the application domain. Therefore, it is difficult to recognize the pattern instance when it is composed with others in a particular design. For this type of pattern, it is essential to show the pattern name and the role played by each element (class, attribute and method) in the instantiation.

However, a domain-specific pattern is instantiated in the scope of a domain. Therefore, it is easy to track the use of the pattern even after it is applied or composed with other patterns. We assume that omitting both the name and the role of pattern attributes and operations will not create any ambiguity. For this reason, we propose to present only the pattern name and the role names of the classes in order to avoid overloaded models. In fact, pattern-related information should be minimized in the class and sequence diagrams for readability [8].

We propose definition of three stereotypes for the explicit visualization of patterns in an application class diagram and sequence diagram:

- <<***patternClass***>> stereotype: It is applied to the *Class* UML metaclass in order to indicate that it is an instantiated pattern class and not originally defined by the designer. We propose definition of two properties related to this stereotype:
  - *patternName* tag : indicates the pattern name.
  - *participantRole* tag : indicates the role played by the class in a pattern instance.
- <<***patternLifeline***>> stereotype: It is applied to the *Lifeline* metaclass in order to distinguish between the objects instantiated from the pattern sequence diagram and those defined by the designer. This stereotype has the same properties as <<patternClass>> stereotype.
- <<***patternInteraction***>> stereotype: It is used to denote an interaction instantiated from the pattern sequence diagram. It has the same properties as <<patternClass>> stereotype.

These stereotypes allow elimination of any confusion when patterns are composed. That is, when two or more classes represent the overlapping part of the composition, the <<***patternClass***>> stereotype shows the roles that these classes play in each pattern.

3.3 UML Extensions for Modeling RT Aspects

In addition to the above described stereotypes distinguishing the fixed parts from the optional and variable parts in the pattern, the specification of RT design patterns needs UML extensions supporting the modeling of RT aspects. Thus, we import stereotypes from HLAM (high level application modeling) and NFP (nonfunctional properties) sub-profiles of MARTE [15]. Note that MARTE provides support required from specification to detailed design of RT embedded systems characteristics. However, only the extensions describing RT application features at a high level of abstraction are taken into account since RT patterns can be instantiated to model many RT applications and not only the embedded systems. From HLAM sub-profile, we import stereotypes modeling quantitative features, such as deadline and period, as well as qualitative features related to behavior, communication, and concurrency:

- <<***rtFeature***>> (real-time feature) stereotype: It allows modeling temporal features. This stereotype extends the metaclasses: message, action, signal, and behavioral features. It possesses nine tagged values among which are: utility (i.e., specification of importance features), relD1 (i.e., specification of a relative deadline), tRef (i.e., time reference used for relative timing properties), absD1 (i.e., specification of an absolute deadline), Miss (i.e., percentage of acceptance for missing the deadline), occKind (i.e., specification of the type of event:

periodic, aperiodic, or sporadic), priority (i.e., specification of priority), boundDl (i.e., relative deadline) and rdTime (i.e., minimal ready time).

- <<***ppUnit***>> (protected passive Unit) stereotype: It is used to model the shared data requiring the specification of concurrency policy. Protected passive units specify their concurrency policy either globally for all their provided services through their concPolicy attribute, or locally through the concPolicy attribute of <<RtService>> stereotype.
- <<***rtUnit***>> (real-time Unit) stereotype: It models a real-time unit that may be seen as an autonomous execution resource, able to handle different messages at the same time. A real-time unit can manage concurrency and real-time constraints attached to incoming messages.
- <<***rtService***>> (real-time service) stereotype: It is used to specify the services concurrency policy (reader, writer, or parallel) provided by real-time units and protected passive units. Besides, the exeKind attribute of the <<rtService>> stereotype specifies how to handle the execution of these services. The exeKind attribute may have the value deferred (i.e., the message is saved in a queue of the unit), local immediate (i.e., the execution is done in the context of the unit receiving the message), or remote immediate (i.e., the execution is done in the context of the calling unit).

From the NFP Modeling sub-profile of MARTE, we import two stereotypes: <<***Nfp***>> and <<***NfpType***>>. The first one extends the Property metaclass. It shows the attributes that are used to satisfy nonfunctional requirements. The second stereotype extends the DataType metaclass. There is a set of pre-declared *NFP_Types* which are useful for specifying NFP values, such as NFP_Duration, NFP_DataSize, and NFP_DataTxRate.

Figure 3 represents the meta-model of UML-RTDP profile. It shows the proposed stereotypes and their base classes. It shows also the relations between the UML meta-classes in order to facilitate the understanding of OCL rules. These rules represent a powerful mechanism for constraining dependencies between pattern variable elements. They are explained in the next section.

4 Definition of OCL Constraints

The introduction of variability using new stereotypes improves genericity but can generate some inconsistencies (e.g., if a mandatory subclass specializes an optional super class, the resulting model is incoherent). Thus, in order to ensure RT design patterns consistency, we propose delimiting the impact of variable elements through the definition of OCL constraints. These latter can be applied to structural and behavioral views of design patterns.

C1: For each optional operation belonging to a pattern class, there must be a corresponding event enclosed in a combined fragment having the "opt" interaction

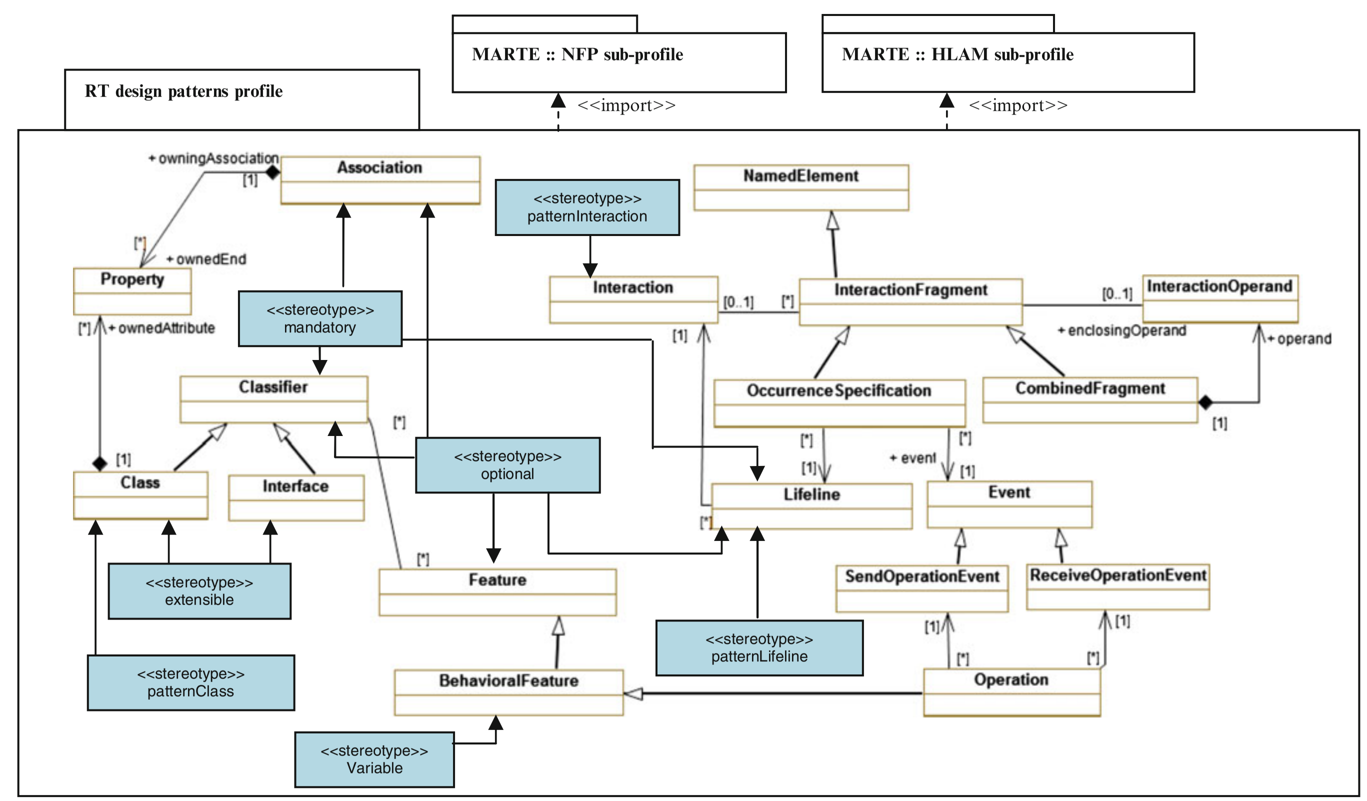


Fig. 3 UML-RTDP profile meta-model

operator. The call of an operation in a sequence diagram is defined by two events: an event generated when invoking an operation and an event generated when receiving an operation.

Note that the method *isStereotyped(S)* used in the following rules is an auxiliary operation indicating if an element is stereotyped by a string S. It is formalized in [18] using OCL as follows:

```
Context Construct::Class::isStereotyped(s: string):Boolean
  isStereotyped = self. extensions ->
  exists(E |E. ownedEnd. type. name =s)
```

```
Context operation inv:
   self. isStereotyped ('optional') or
   self. Class. isStereotyped ('optional')
  implies
  self. sendOperationEvent ->
  forall (e | e. occurrenceSpecification ->
  forall ( I | I.enclosingOperand.combinedFragment.
  InteractionOperator = InteractionOperatorKind::opt))
  and
  self. receiveOperationEvent ->
  forall (e | e. occurrenceSpecification ->
  forall ( I | I.enclosingOperand.combinedFragment.
  InteractionOperator = InteractionOperatorKind::opt))
```

Figure 4 shows an example of an optional operation in the *Controller* class. This operation must be enclosed in a combined fragment having the "opt" interaction operator in the sequence diagram.

C2: Each association, which is related to an optional class, must be stereotyped <<optional>>.

```
Context class inv:
  self. isStereotyped ('optional')
  implies
  self. ownedAttribute -> forall(a | a.owingAssociation.
  isStereotyped ('optional'))
```

C3: Each class, which inherits the feature of an optional super class, must be stereotyped <<optional>>.

```
Context redefinableElement inv:
  self. isStereotyped ('optional')
  implies
  self. redefinedElement. isStereotyped ('optional')
```

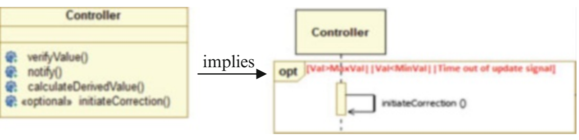


Fig. 4 Example of an optional operation

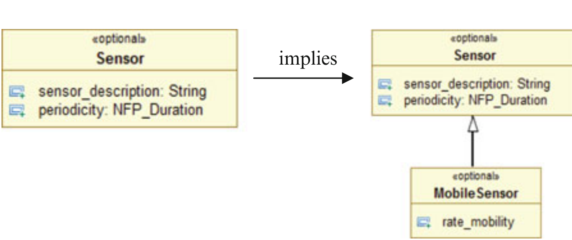


Fig. 5 Example of an optional class

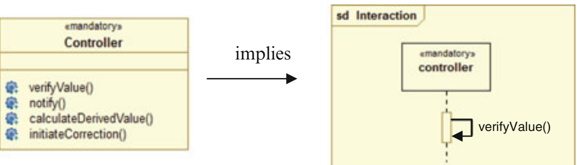


Fig. 6 Example of a mandatory class

In Fig. 5, we present an example of an optional superclass Sensor. In this case, the MobileSensor class must be stereotyped <<optional>> since it specializes the Sensor class.

C4: Each class, which implements an optional interface, must be stereotyped <<optional>>.

```
Context namedElement inv:
  Let realizationLink set (realization) =
  self. clientDependency ->
  select (c | c. oclIsKindOf (realization)

  in

  realizationLink. supplier ->
  forall (I | I. isStereotyped ('optional')

  implies

  self. isStereotyped ('optional')
```

C5: Each class, which is stereotyped <<mandatory>>, must have a corresponding object in a sequence diagram stereotyped <<mandatory>>.

```
Context interactionFragment inv:
  self. oclIsKindOf (occurrenceSpecification) and
  self. event. oclIsKindOf (callEvent) and
  self.event.operation.class.isStereotyped ('mandatory')
  implies
  self. lifeline. isStereotyped ('mandatory')
```

To illustrate the C5 constraint, let us consider the example shown in Fig. 6 where the controller is a mandatory class, and as a consequence, it must have a corresponding object stereotyped <<mandatory>> in the sequence diagram.

5 A Real-Time Design Pattern Example

In this section, we illustrate the UML-RTDP extensions through an example of a reusable RT design pattern which is the controller pattern. This pattern aims to model both the control of data acquired from the environment and the initialization of corrective action(s) in case of constraint violation. The creation of the controller pattern is based on the application of RT design pattern development process. This process allows the generation of RT patterns from a set of concrete application designs. It defines unification rules that apply a set of comparison criteria on various applications in the RT domain. In addition, domain requirements and constraints are extracted and then confronted to the generated patterns, in order to validate them.

5.1 RT Controller Pattern Specification

RT applications perform several RT processes among which are: the RT data acquisition and the data control processes. We mainly focus in this chapter on modeling the static as well as the dynamic view of the RT data control process through the definition of the controller pattern.

- **Interface:**
  Name: controller pattern
  Context: This pattern is applicable in all RT applications which need to be managed by real-time database (RTDB) systems. In fact, not only an RTDB has all the requirements of traditional databases, but it also requires management of time-constrained data and time-constrained transactions [20].
  Intention: The pattern aims to model the control of the data acquired from the environment and the initialization of corrective action(s) if a violation is found.
- **Solution:**
  Static specification: Figure 7 presents the controller pattern structural view. This pattern has three participants: the observed element, the controller, and the operator.

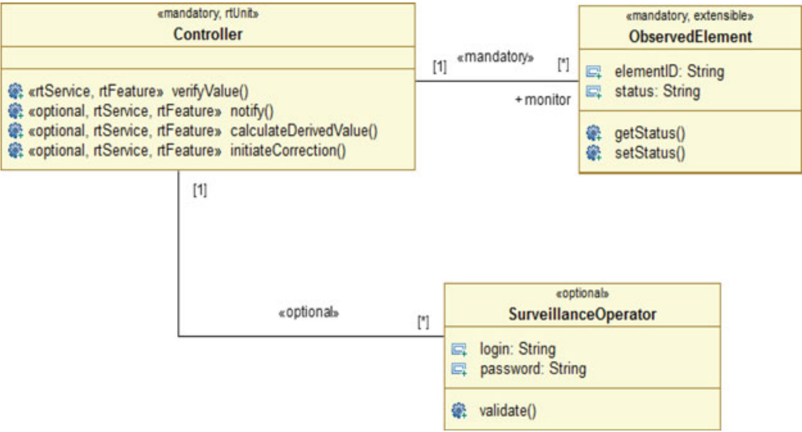


Fig. 7 RT controller pattern structural view

Participants

- Observed_element: This class represents the description of a physical element that is supervised by the controller. It can be an aircraft, a car, a road segment, and so on. One or more measure types (i.e., temperature, pressure, etc.) of each observed element could determinate its evolution. The *ObservedElement* class has the *ElmentID* and *Status* fundamental attributes. In addition, it has the *SetStatus ()* method allowing the updating of the status of an observed element according to the variation of the captured values.
- Controller: The controller has to monitor physical elements for responding to conditions that might violate safety. It is the main class which coordinates the other classes. It represents an active entity that has the capacity to handle simultaneously different messages for the check of the system state. Consequently, the controller class is stereotyped <<rtUnit>>.

There are two ways to update the state of observed elements: pull mechanism and push mechanism. In the pull mechanism, the controller takes periodically the values captured for each observed element. Then, it updates each measure's value and checks if it is between the minimum value and the maximum value that define the interval for which the controller does not detect an anomaly. If a captured value does not satisfy the boundary constraint, then the controller initiates some corrective actions, such as a reset and a shutdown, or sends an alarm to notify an operator.

In the push mechanism, the controller receives periodically a signal to be notified about the data that must be updated for each observed element. In this case, the controller is waiting for a signal. If this signal does not arrive on time, then the controller performs appropriate recovery actions [21].

As illustrated in Fig. 7, the controller class has four methods. The only fundamental method is *VerifyValue()* since it is essential to check that the boundary constraints are fulfilled for all RT applications. This method is performed periodically. In

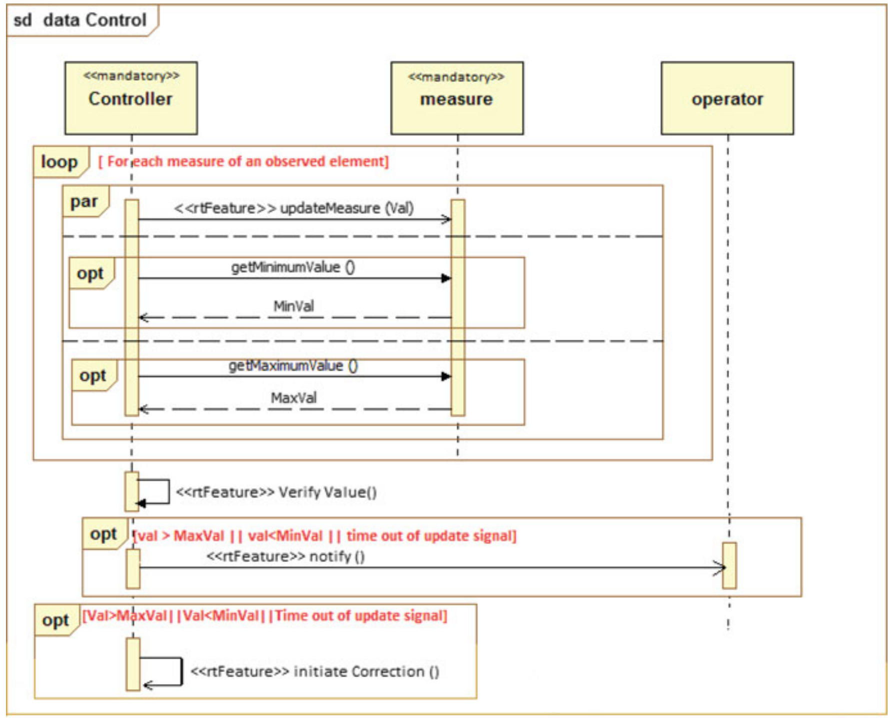


Fig. 8 RT controller pattern behavioral view

addition, it must be achieved before a deadline. Thus, the *VerifyValue()* method is stereotyped <<rtFeature>>. The periodicity, the relative deadline and the absolute deadlines of this method are defined, respectively, through *period*, *relDl* and *absDl* attributes of the <<rtFeature>> stereotype. Similarly, the *CalculateDerivedValue()* method is stereotyped <<rtFeature>> since it is sporadic and has to meet the deadline defined by the designer. It is represented as an optional method since it can be omitted when the designed application does not have derived measures (i.e., calculated from measure's value captured by sensors). The methods *notify()* and *initiateCorrection()* are optional since the choice of the appropriate recovery action depends on the application instantiating the pattern.

Besides, the <<rtService>> stereotype is applied to all the methods of the controller class because the concurrency policy must be specified for each method having timing constraint and belonging to an RT unit.

- Operator: The operators supervise the alarm signals sent by the controller. They provide decisions to validate reported incidents in case the controller only reports errors and does not have the responsibility of initiating corrective actions; or in case the confirmation of an operator is needed to achieve the correction.

Dynamic specification: Figure 8 presents the controller pattern behavioral view. It describes the interactions between objects to update each measure's value and to verify that each measured value is in the closed range [Minimum-Value, Maximum-value]. If this constraint is violated or the update measure signal received by the controller occurs too late, then the controller notifies the operator or initiates the appropriate recovery actions.

The controller pattern behavioral view is composed into a set of combined fragments using interaction operators. We use three fundamental operators: *loop*, *par*, and *opt*. The *loop* operator specifies an iteration of interactions. The *par* operator designates that the combined fragment represents a set of parallel interactions that are performed simultaneously. Finally, the *opt* operator designates that the combined fragment represents an optional behavior that can be either performed or not performed. Note that the optional combined fragments shown in Fig. 8 represent interactions that are relative to the optional methods of controller pattern class diagram.

5.2 RT Controller Pattern Instantiation

As shown previously, an RT design pattern contains a set of variation points. Therefore, to derive a specific RT application model through pattern instantiation, some choices associated with these variation points are needed. The refinement of a pattern model and the choice of the appropriate optional elements constitute the first step of pattern instantiation. The second step is based on model transformation. It automatically generates the application model deploying the instantiated pattern.

In this section, we illustrate the instantiation of the RT controller pattern to design the COMPASS system [22], which represents an example of a freeway traffic management system. We focus precisely on modeling the compass control data subsystem and we explain how this design issue can be facilitated by the reuse of the RT controller pattern.

First, the fundamental elements of the pattern are instantiated. Thus, the *Controller* and *Observed_element* classes are instantiated, respectively, by *TrafficController*, *Vehicle*, and *RoadSegment* classes (cf. Fig. 9). The vehicles and road segments represent the physical elements that are supervised by the controller.

Then, the controller pattern optional elements are instantiated. Figure 9 shows that the *Operator* optional class is instantiated since it is essential to notify the operators of any detected events in the COMPASS system.

Finally, the specific elements relative to the freeway traffic management application are added: the *Incident* class and the attributes *startPointLocation* and *endPointLocation* of *RoadSegment* Class.

The instantiation of the RT controller pattern behavioral view is represented in Fig. 10. The figure shows that the optional combined fragment relative to the initiation of corrective actions is omitted since the method *initiateCorrection()* is not instantiated in the corresponding application class diagram.

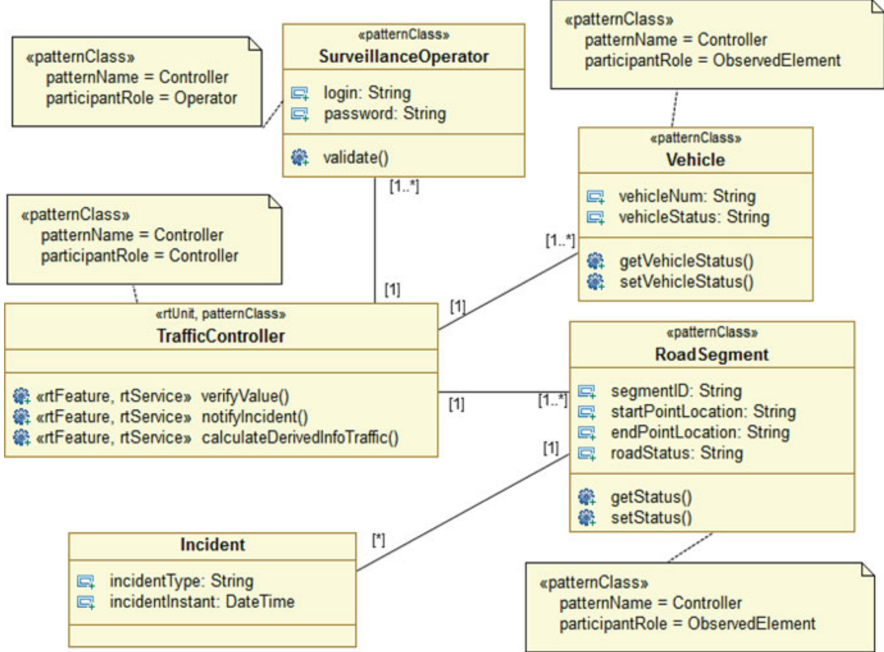


Fig. 9 Instantiation of a controller pattern (structural view)

6 Tool Support for UML-RTDP Profile

We have developed a toolset supporting our UML-RTDP profile using the Papyrus plug-in [23] of Eclipse platform [24]. In fact, Papyrus provides extensive support for UML profiles. It includes all the facilities for defining and applying UML profiles in a very rich and efficient manner. It allows also OCL constraint specification and checking.

When designing our profile, we have customized the UML2 class and sequence diagrams by adding the relevant stereotypes defined in the UML-RTDP profile. We have specified OCL constraints for UML element at the profile meta-model level in order to verify variation points coherence. These constraints are checked against pattern models.

Then, we have imported the Papyrus plug-in that implements the OMG specification of MARTE profile in order to use the appropriate stereotypes modeling quantitative and qualitative features of RT systems.

Finally, we have embedded the UML-RTDP profile within an Eclipse plug-in in order to simplify the manipulation of the stereotypes and their related properties and to ensure better compatibility and extensibility.

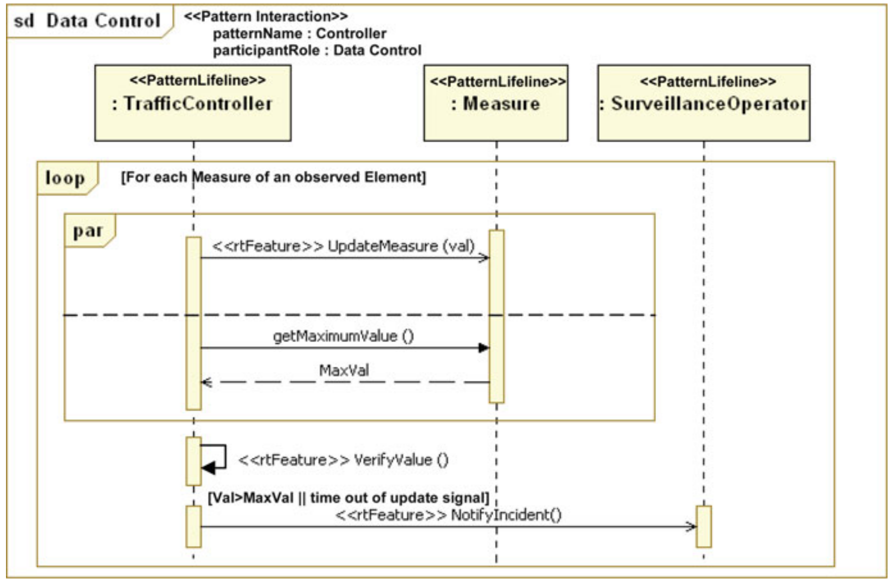


Fig. 10 Instantiation of a controller pattern (behavioral view)

7 Conclusion

RT applications design is a delicate task since these applications must take into consideration RT constraints and also since it is an evolving domain. Thus, it is necessary to benefit from developers' previous experiences and to profit from existing reuse techniques such as patterns.

In this chapter, we have proposed UML-based extensions for RT design patterns representation. These extensions concern UML class diagrams and UML sequence diagrams. They help the designer determine the variable elements that may differ from one application to another and allow identifying, easily, design patterns when they are applied to model a particular RT application. We have, also, proposed some OCL constraints that may be seen as well-formedness rules for modeling RT design patterns.

Besides, this chapter proposed guiding the designer in modeling features specific to the RT domain through the use of stereotypes imported from MARTE profile. Moreover, it illustrated the proposed notation through the specification of an RT controller pattern and its instantiation to design a freeway traffic management system.

In our future efforts, we will improve the UML-RTDP profile with the definition of other constraints. We will also focalize on the implementation of RT design patterns instantiation. For this purpose, we have defined a set of transformation rules in order to add more assistance when generating application models by

reusing patterns. This could bring new benefits and impetus for both the knowledge capturing techniques and the software development process quality.

References

1. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Edition, Reading, MA (1994)
2. Port, D.: Derivation of Domain Specific Design Patterns. USC Center for Software Engineering, Los Angeles, CA (1998)
3. OMG: UML 2.0 OCL specification (2003)
4. Eden, A.H., Gil, J., Hirshfeld, Y., Yehudai, A.: Towards a mathematical foundation for design patterns. Technical Report, Dept. of information technology, U. Uppsala (1999)
5. Mikkonen, T.: Formalizing design patterns. In: Proceedings of the 20th International Conference on Software Engineering—ICSE, pp. 115–124 (1998)
6. Bouassida, N., Ben-Abdallah, H.: Extending UML to guide design pattern reuse. In: Proceedings of the 6th Arab International Conference On Computer Science Applications, Dubai (2006)
7. Yacoub, S.M., Ammar, H.: Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems, Addison-Wesley Edition, Reading, MA, August 2003
8. Dong, J., Yang, S., Zhang, K.: Visualizing design patterns in their applications and compositions. J. IEEE Trans. Software Eng. **33**(7), 433–452 (2007)
9. Arnaud, N., Front, A., Rieu, D.: Expression et usage de la variabilité dans les patrons de conception. Revue des sciences et technologies de l'information, série: Ingénierie des Systèmes d'Information **12**(4), 21–24 (2007)
10. Loo, K.N., Lee, S.P.: Representing design pattern interaction roles and variants. Proceedings of the 2nd International Conference on Computer Engineering and Technology (ICCET' 2010), pp. 470–474 (2010)
11. Reinhartz-Berger, I., Sturm, A.: Utilizing domain models for application design and validation. Inform. Software Technol. **51**, 1275–1289 (2009)
12. Douglass, B.: Real Time UML, Advances in The UML for Real-Time Systems. Pearson Education Inc, Boston, MA (2004). 0-321-16076-2
13. Lanusse, A., Gérard, S., Terrier, F.: Real-time modeling with UML: the ACCORD approach. In Bézivin J., Muller P.-A. (eds.) The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998
14. OMG: UML Profile for Schedulability, Performance and Time, v1.1, formal/2005-01-02 (2005)
15. OMG: A UML profile for MARTE: modeling and analysis of real-time embedded systems, OMG document number: ptc/2008-06-09 (2008)
16. Idoudi, N., Louati, N., Duvallet, C., Bouaziz, R., Sadeg, B., Gargouri, F.: How to model a real-time database. In: Proceedings of the 12th IEEE International Symposium on Object-oriented Real-time distributed Computing (IEEE ISORC'2009), pp. 321–325. Tokyo, Japan, 17–20 March 2009
17. OMG: Unified Modeling Language (UML) infrastructure: v2.1.2, formal/2007-11-04 (2007). Accessed April 2013
18. Ziadi, T., Jézéquel, J.-M., Fondement, F.: Product line derivation with UML.In: Proceedings Software Variability Management Workshop, Univ. of Groningen Department of Mathematics and Computing Science, February (2003)
19. Rekhis, S., Bouassida, N., Duvallet, C., Bouaziz, R., Sadeg, B.: a process to derive domain-specific patterns: application to the real time domain. Proceedings of the 14th International Conference on Advances in Databases and Information Systems (ADBIS'2010), LNCS 6295, pp. 475–489. September 2010

20. Ramamritham, K., Son, S., DiPippo, L.: Real-time databases and data services. Real-Time Syst. **28**, 179–215 (2004)
21. Douglass, B.P.: Real-Time Design Patterns: Robust Scalable Architecture for Real Time Systems, Addison-Wesley Edition, Reading, MA, 27 September 2002
22. COMPASS Website. http://www.mto.gov.on.ca/english/traveller/compass/main.htm. Accessed April 2013
23. Papyrus UML2 tool version 1.11. http://www.papyrusuml.org (2007)
24. Eclipse Platform Version 3.4.2, http://www.eclipse.org/platform/. Accessed April 2013

When Aspect-Orientation Meets Software Product Line Engineering

Iris Reinhartz-Berger

Abstract Aspect-oriented software development (AOSD) and software product line engineering (SPLE) are two approaches for software reuse, which promote model-driven development and variability management. While AOSD supports developing crosscutting concerns separately from traditional units and weaving them to different software products, software product line engineering (SPLE) handles the development and maintenance of families of software products utilizing different domain and application engineering techniques. In this chapter, we review the existing points of synergy between these two approaches and, in particular, the complementary and aggregative use of these approaches. Furthermore, we present a method that uses aspect-oriented principles for horizontal reuse and domain engineering guidelines for vertical reuse. We term this kind of use *dimensional synergy*. The presented method supports defining families of aspects and their weaving rules applied to families of software products, potentially increasing the reuse throughout the entire development life cycle. We exemplify the method on a Check-In check-Out product line and a family of security aspects, utilizing UML 2 class and sequence diagrams.

Keywords Aspect-orientation • Domain analysis • Domain engineering • Early aspects • Software product line engineering • UML • Variability

1 Introduction

The significant increase in systems complexity and variety in the last decades caused software engineering to develop different reuse approaches [17]. Some of these approaches support division and decomposition of complex problems into

I. Reinhartz-Berger (✉)
Department of Information Systems, University of Haifa, Haifa 31905, Israel
e-mail: iris@is.haifa.ac.il

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_4, © Springer-Verlag Berlin Heidelberg 2013

smaller ones that may be solved one at a time with relatively simple means. They further suggest how to gather and integrate these solutions into holistic ones that solve the complex problems at hand. *Aspect-oriented software development* (AOSD) [16, 31] is an example of an approach in this category. It aims to provide a way of modularization according to which crosscutting concerns are separated from traditional units during the entire software development life cycle. AOSD treats these crosscutting concerns as *aspects* that can be *woven* into software systems in order to fulfill the requirements at hand. AOSD originated in programming and implementation [16, 52] and has been percolated to early development phases, i.e., requirements analysis, architecture design, and detailed design stages [9, 15]. In early development stages, aspects cannot be localized and tend to be scattered over multiple early life cycle modules. If early aspects are not effectively modularized, it is not possible to reason about their effect on the system or on each other. Thus, different methods that deal with aspects during the analysis and design phases have been proposed (e.g., [23, 28, 34, 53, 59]). Nevertheless, despite a stable notion of aspects at the programming level, aspects and their representation in early development stages have not been consolidated yet and a standard has not emerged.

Other reuse approaches generalize possible solutions and enable their usage in different contexts. An example of an approach in this category is *software product line engineering* (*SPLE*) [13, 42]. SPLE handles the development and maintenance of families of software products. For these purposes, *core assets*, also known as *domain artifacts*, are defined as parts that are built to be used by more than one product in the family, while *product artifacts*, or *application artifacts*, are specific parts of the software products. Accordingly, SPLE activities are divided into domain and application engineering: *domain engineering* supports the development and maintenance of software product line artifacts, i.e., core assets, whereas *application engineering* mainly deals with the adaptation and customization of core assets in order to develop particular applications and software products. In order to be reusable and suitable to a wide variety of products, the development of core assets has to be based on commonality and variability analysis [6, 50, 56]: *commonality* refers to the kernel of the software product line that is reused by all the members of the family, and *variability* is the ability of a core asset to be efficiently extended, changed, customized, or configured for use in a particular software product. The main corpus of SPLE methods utilizes feature-oriented or UML-based notations for specifying common and variable aspects [8]. Feature-orientation supports the specification of core assets as sets of characteristics relevant to some stakeholders and the relationships and dependencies among them [29]; UML-based methods commonly suggest profiles for handling core assets specification and especially variability-related issues [25].

Being both model-driven and tackling similar problems, points of synergy between AOSD and SPLE have recently been explored. In particular, two main types of synergy can be identified: (1) the two approaches are *complementary*—SPLE, and especially feature-oriented methods, are used in early development stages, while AOSD principles are utilized for designing architectures and for implementation purposes, e.g., [4, 20, 36, 37]; (2) the two approaches are *aggregative*—AOSD

principles are used for supporting different SPLE activities, such as requirements engineering [38] and variability representations [39, 55]. Additional types of relationships between the two approaches also exist in the literature, but these types usually refer to AOSD and SPLE as two approaches that are meant to be utilized for different purposes. Barth et al. [7], for example, claim that the two approaches contribute to generative software development, but SPLE is the key to achieve systematic software reuse, while AOSD provides "better separation of concerns and composition mechanisms."

In this chapter, we concentrate on complementary and aggregative combination of AOSD and SPLE and review studies in these categories. We further introduce the notion of *dimensional synergy*, according to which SPLE is used for vertical reuse (from core assets to product artifacts) and AOSD is utilized for horizontal reuse (namely introducing crosscutting concerns to core assets or product artifacts). This chapter also presents a method that follows the dimensional synergy principles. This method extends a domain engineering approach, called *Application-based DOmain Modeling (ADOM)* [44, 45, 51], with the aim of representing domain aspects and their weaving rules applied to software product lines. ADOM is used as a framework for defining aspect, base, and woven models at different abstraction levels, i.e., product (or application) and product line (or domain) levels.

The remainder of this chapter is organized as follows. Section 2 reviews AOSD and SPLE fields, as well as their existing types of synergy. Section 3 introduces the principles of dimensional synergy, while Sect. 4 elaborates on the ADOM-based method. For demonstrating the method, UML 2 class and sequence diagrams are utilized on a Check-In Check-Out (CICO) product line and a family of security aspects. Finally, Sect. 5 summarizes, discussing the expressiveness of the method and its potential to increase reuse, and refers to future research directions.

2 Literature Review

2.1 Aspect-Oriented Software Development (AOSD)

As noted, much work has been done for percolating aspect-oriented programming (AOP) notions to early development phases. These studies are mainly model-driven and offer ways to specify and represent aspects and to guide their weaving into particular systems or software products. Most studies utilize UML or its extensions for these purposes and distinguish between base models and aspect models. *Base models* represent software systems or applications and are the target of the weaving process, while *aspect models* specify crosscutting concerns and are the objects on which the weaving process is performed. Table 1 summarizes the characteristics of different AOSD modeling methods, focusing on their specification means and weaving capabilities (i.e., the weaving process and conditions). As can be seen, most of the listed methods deal with aspects at a specific abstraction level, partially

Table 1 A comparison of the reviewed aspect-oriented methods

Method name	Declared goals	Specification means	Weaving process	Weaving conditions
Theme/UML [10]	Providing visual representation to Theme/Doc outcomes	Template arguments and merging, override and binding mechanisms	Weaving is done through merge and override relations within the aspect model	
Katara and Katz [30]	Dealing with aspect compositions and interactions	A set of stereotypes and tags for defining different types of binding	No separation between the concerns and the weaving rules is made	The aspect can be woven only to base models that fulfill its required part
PCS [28]	Providing means for implementing concern-oriented software architectures	AspectJ syntax	No separation between the concerns and the weaving rules is made	
Groher and Voelter [22]	Weaving models and meta-models	Models and meta-models in the Eclipse Modeling Framework	No separation between the concerns and the weaving rules is made	
AODM [54]	Modeling AspectJ style programs with UML	AspectJ syntax	No separation between the concerns and the weaving rules is made	
Aldawud et al. [1, 2]	Visualizing and documenting aspect-oriented software artifacts	Stereotypes, class diagrams, and statecharts	No separation between the concerns and the weaving rules is made	
UFA [27]	Designing aspects emphasizing reuse issues	Callin and callout bindings	Connector packages "connects" aspect and base models	
AML [23, 24]	Modeling AspectJ style programs with UML	AspectJ syntax	Connector packages "connects" aspect and base models	

(continued)

Table 1 (continued)

Fuentes and Pablo [18]	Executing aspect-oriented models	A UML profile and an XMI weaver	Pointcut packages are defined; the focus is on weaving behavioral aspects	
Klein et al. [33]	Weaving multiple behavioral aspects	Four types of sequence diagrams, each of which represent a different type of joint point	Weaving is specified in separate sequence diagrams	
Reddy et al. [43]	Composing aspects based on their signature	Class diagram element signatures and a composition meta-model	No separation between the concerns and the weaving rules is made	
JPDD [53]	Visualizing the selection criteria that an aspect requires from a base model	Special kind of diagrams, called join point designation diagrams (JPDD)	Does not support weaving	Query models are defined in the JPDD notation

handling weaving guidelines (or completely neglecting them). Those methods that present weaving guidelines do that very closely to the implementation and programming level, falling short in considering the entire spectrum of modeling concepts not present in programming languages [49]. They further refer to aspects as particular concerns and, hence, suggest different ways to weave them to specific systems. In particular, most of the listed methods do not specify the conditions and the situations to which an aspect is appropriate and those that support this kind of specification do not use them for guiding the weaving process.

2.2 Software Product Line Engineering (SPLE)

Many SPLE methods have been introduced for specifying common and variable elements of software product lines. The main aids for specifying common elements are: (1) multiplicity, which primarily differs between mandatory elements that identify the product family and all products that belong to the family must include them, and optional elements that may add some value to the product, when selected,

but not all the products that belong to the family will include them; and (2) (inter-) dependencies, which are restrictions for selecting groups of optional elements, e.g., an element *requires* or *excludes* another element. The possible variability in a product line is primarily specified in terms of: (1) variation points, which identify locations at which variable parts may occur; (2) variants, which realize possible ways to create particular product artifacts at certain variation points, (3) rules for selecting variants, e.g., via specifying cardinalities which define the minimal and the maximal numbers of variants that have to be selected in a given variation point; and (4) guidelines for adding product-specific variants, e.g., via specifying open vs. closed variation points that, respectively, allow or do not allow the addition of product-specific variants.

As noted, the majority of SPLE methods can be classified as feature-oriented or UML-based.[1] The primary way for specifying core assets in feature-oriented methods is via feature diagrams, which are basically trees of features [29]. Each node in a feature diagram represents a feature, whereas the tree's root represents the main concept of the product line. Features can be decomposed into sub-features and the edges represent dependencies between features, including mandatory vs. optional features, alternatives, and OR features. Most feature-oriented methods are based on or extend feature-oriented domain analysis (FODA) [29].

The second largest category of SPLE methods is based on UML [8]. These methods commonly define profiles for supporting variability-related issues. Sometimes a variability model is introduced orthogonally to the UML models, for instance in [42]. Since UML is utilized both in SPLE and in AOSD, we will elaborate a little bit more on UML-based methods in this context.

Table 2 summarizes several such methods according to their ability to specify common and variable elements. As can be seen, commonality-related issues are usually specified using dedicated stereotypes for differentiating mandatory (sometimes called kernel) and optional elements, although these stereotypes are usually only associated with variation points and variants and not with other elements in the core assets. Several works explicitly refer to dependencies between elements in the form of «alternative_or», «alternative_XOR», «requires», and «mutux» stereotypes.

All the surveyed studies refer to variability, usually using stereotypes such as «variation», «variation point», or «V» for specifying variation points and stereotypes such as «variant» or «variable» for modeling variants. Other methods explicitly specify only one of these concepts and the other is implicitly specified by its relationships with the modeled concept (commonly the relationship between these concepts is specified via inheritance relations or dependencies).

Regarding variant selection rules, most methods that refer to this criterion support only "OR" (0..*) and "XOR"(1..1) selections and do not support range selections. Some methods allow wider representation of selection rules by using tagged values, e.g., Clauß and ADOM in Table 2, and variation point attributes, e.g., SPLIT in

[1] A few methods can be classified as both, e.g., [21]. However, these methods usually focus on one of these paradigms and extend the methods towards the other.

Table 2 Partial comparison of UML-based Methods (adapted from [46])

	Commonality		Variability		
Method name	Multiplicity	Inter-dependencies	VP and variants	Variants selection (at VPs)	Addition of variants
PLUS [19]	Mand.—«kernal» Opt.—«optional»	NS	VP—NS Var—«variant»	By default, variants are mutually exclusive selected	NS
Halmans and Pohl [26]	Only for VPs: Mand.—filled triangle Opt.—unfilled triangle	NS	VP—triangle Var—«variant»	According to the triangle color and the relationship type (to variants)	NS
Ziadi et al. [60]	Mand.—default Opt.—«optional»	Generic constraints	VP—«variation» Var—«variant»	NS	NS
Alves de Oliveira et al. [3]	only for variants: Mand.—«mandatory» Opt.—«optional»	«Requires», «mutux» (between variants)	VP—«variation point» Var—elements associated to VPs	«alternative_or», «alternative_XOR”	NS
Morisio et al. [40]	NS	NS	VP—«V» Var—inheritance to VP	The default is OR, «xorV» specifies mutual exclusiveness	NS
Robak et al. [48]	NS	NS	VP—NS Var—«variable» + tagged values	Specified via {or} and {xor} constraints on the associations	NS
SPLIT [14]	Only for VPs: The existence attribute indicates whether the VP is optional or mandatory	NS	VP—«variation point» Var—elements connected to VPs via associations	Through attributes of the VPs	Through attributes of the VPs
VPM [57]	Only for VPs: Mand.—m Opt.—o	NS	VP—vp V—«variant»	NS	Through callbacks and guidelines

(continued)

Table 2 (continued)

	Commonality		Variability		
Method name	Multiplicity	Inter-dependencies	VP and variants	Variants selection (at VPs)	Addition of variants
Clauß [11, 12]	Mand.—by default Opt.—«optional»	«Requires», «mutex»	VP—«variation point» Var—«variant»	Through the multiplicity tagged value	NS
Riebisch et al. [47]	NS	NS	VP—NS Var—«variant»	NS	NS
ADOM [44, 45, 51]	Mand.—«mandatory» Opt.—«optional»	«Requires», «excludes»	VP—«variation point» Var—«variant»	Through the card tagged value	Through the open tagged value

NS not supported, *Mand* mandatory elements, *Opt* optional elements, *VP* variation point, *Var* variant

Table 2. Only a few methods (e.g., SPLIT, VPM, and ADOM in Table 2) refer to addition rules, i.e., enable specifying if new variants can be added to variation points.

2.3 AOSD and SPLE: Points of Synergy

Different studies have researched the relationships between AOSD and SPLE, introducing methods that rely on principles from both fields. These methods can be roughly divided into methods that refer to the two fields as complementary and methods which aggregately use AOSD and SPLE principles for the same purposes.

Methods in the first category mainly use SPLE for early development stages, while AOSD is used for later phases of detailed design and implementation. Lopez-Herrejon and Batory [37], for example, suggest emulating function composition in AOP using a small set of advice, bounded quantification, and algebraic specification. Lee et al. [36] offer combining feature-oriented analysis (FOA) and AOP in order to benefit from the "essential design drivers" provided by FOA and the AOP's "effective mechanisms for encapsulating crosscutting abstractions into modular units and integrating the units without changing the rest of the system." Griss [20] outlines a practical development process that integrates feature-driven domain analysis and design and aspect-oriented implementation techniques, in order to better structure models, designs, and code. Anastasopoulos and Muthig [4] examine whether AOP can serve as a product line implementation technology and to what extent. They conclude that in this context AOP is especially suitable for supporting variability across several components, whereas its suitability for supporting variability inside of single components requires further investigation.

Methods in the second category, which support aggregative synergy, use both AOSD and SPLE principles for various development tasks. Using ADORA, for example, Stoiber et al. [55] propose visualizing and modeling variability using aspect-orientation and table-based modeling of configuration possibilities and constraints. Morin et al. [39] argue that aspect-oriented modeling can help users design optional and variant parts of a model. They further claim that the ability to weave aspects incrementally into base models enables constructing final products step-by-step. Their generic approach supports generating target languages and some weaving instructions to any given meta-model. After deriving an aspect by choosing the most appropriate variants and options, aspect configurations can be woven into base models, to integrate new features and propose different variants of the system. Kulesza et al. [35] allow for improved customization and instantiation of frameworks by using crosscutting feature models. The approach intends to provide guidelines to modularize the implementation of framework features using aspects. Loughran et al. [38] describe an approach for facilitating requirements analysis, commonality and variability analysis, and concern identification, utilizing natural language processing and aspect-oriented techniques.

Reviewing the different studies and especially those mentioned in the current section, we observed that while SPLE refers to two abstraction levels, application and domain engineering, AOSD treats aspects as particular concerns that can be woven in different ways to specific systems or software product lines. However, analyzing the commonality and variability within families of aspects can be beneficial for developing and maintaining aspects. Thus, we propose four weaving possibilities, according to the levels of aspects and the levels of systems or software products to which the aspects are woven. This type of synergy, which we call *dimensional synergy*, is elaborated next

3 Dimensional Synergy of AOSD and SPLE

In order to explain and demonstrate our suggestion for dimensional synergy of AOSD and SPLE, we will use a CICO product line [32] and a family of security aspects. Applications or products in the CICO line manage operations for item enrollment and signing-out. An item can be borrowed by one user at a time and waiting lists for items may be (optionally) maintained. Library, video rental, hotel management, and car rental systems are all CICO applications. These applications may involve different security aspects. Security in the context of computer science concerns risk management trade-offs in the areas of confidentiality, integrity, and availability of electronic information that is processed by or stored on computer systems [58]. Systems which contain fundamental flaws in their security designs cannot be made secure without compromising their usability. However, in many cases security techniques can be woven into (existing) system designs and, hence, may be considered as aspects. Examples of particular security aspects are: (1) authorization which deals with protecting computer resources by allowing those resources to be used only by consumers that have been granted authority to use them, (2) authentication which is the act of establishing or confirming something (or someone) as authentic, that is, that claims made by or about the thing are true, and (3) fraud protection which deals with protecting someone (or something) from fraud activities, such as theft, false billing, and bait, by recording the history of system activities and analyzing the collected data.

Table 3 summarized four types of weaving processes, according to the level of aspects and systems they involve. In the first category, marked as (a) in the table, both system and aspect are at the domain engineering level. This means, for example, that the family of security aspects is woven to the CICO product line in order to create a family of secured CICO applications. In the second category, marked as (b), the family of aspects is woven to a particular system, helping develop, for example, a secured car rental system that involves different security aspects, including authorization, authentication, and fraud protection. The third and fourth categories, respectively, marked as (c) and (d), support weaving specific aspects to either a family of systems (a software product line) or to a particular system (a software product). These ways, the development of authorized CICO applications

Table 3 The meaning of weaving aspects to systems in different abstraction levels

	Level of aspect	Level of system	The meaning of the weaving
a.	Domain engineering	Domain engineering	The result resides at the domain engineering layer. Both system and aspect models should be specialized into a particular system that includes specific aspects
b.	Domain engineering	Application engineering	The result resides at the domain engineering layer. The aspects that belong to the same family and are included in the particular system have to be specialized
c.	Application engineering	Domain engineering	The result resides at the domain engineering layer. A family of system models includes the particular aspect. Each system in the family similarly integrates the particular aspect into its architecture
d.	Application engineering	Application engineering	The result resides at the application engineering layer. The particular system includes the particular aspect. No specialization or further treatments are required

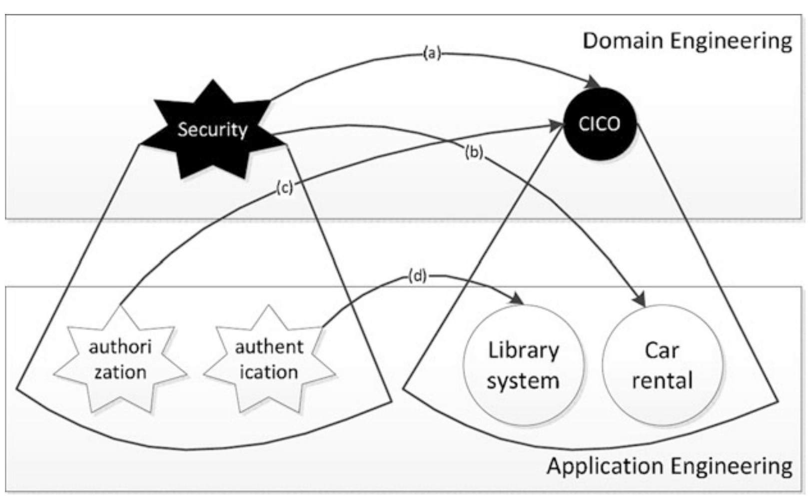


Fig. 1 Possibilities for weaving aspects and systems at different abstraction levels

and the development of a particular authenticated library system are supported. These examples are presented in Fig. 1, where circles represent systems or families of systems and stars represent aspects or families of aspects. The arrows represent weaving processes. This classification calls for utilizing AOSD for horizontal reuse and SPLE, or more accurately domain engineering principles—for vertical reuse.

For supporting these kinds of weaving processes, we propose a method that extends a domain engineering approach, called ADOM [44, 45, 51]. ADOM, which stands for Application-based DOmain Modeling, comprises three layers of

modeling: application, domain, and language. The *application layer* consists of models of particular applications and systems, including their structure and behavior. The *language layer* includes meta-models of modeling languages, such as UML. The intermediate *domain layer* consists of specifications of various application families or product lines, including their common and variable elements. Furthermore, constraints among the different layers are enforced; in particular, the domain layer enforces constraints on the application layer, while the language layer enforces constraints on both the domain and application layers.

Separating the application and domain layers from the language layer, ADOM can be used in conjunction with different modeling languages, but when adopting ADOM with a specific modeling language, this language is used in both application and domain layers, easing the task of application creation (instantiation) and validation by employing the same constructs and terminology in both layers.

ADOM was selected for this work due to the following main reasons: (1) it can be used with UML [41], which is widely employed within both SPLE and AOSD fields; (2) it supports the specification of models in the domain and application layers with similar means; (3) its commonality and variability expressiveness exceeds that of other UML-based SPLE or domain engineering methods (see Table 2). In particular, it explicitly refers to the selection and addition of variants in certain variation points, enables explicit specification of both variation points and variants, and allows specifying ranges of multiplicity and not just mandatory and optional elements.

Next, we elaborate and exemplify the extension of ADOM for increasing the reuse of aspects in both application and domain layers. We call this extension—aspect-oriented ADOM.

4 Aspect-Oriented Application-based DOmain Modeling

Aspect-oriented ADOM supports specification of base and aspect models. A *base model* describes an application, a system, or a family of such (i.e., a software product line). It intend to stand alone and exhibits both structure and behavior. An *aspect model* describes the structure and behavior that characterize a particular concern. It is not intended to stand alone but to be woven into base models. An aspect model can be defined in the application or domain layer, respectively, specifying a specific concern or a family of concerns. An aspect model is divided into concern specification, match pattern, and merge guidance. The *concern specification* deals only with issues that are relevant to the concern at hand. The *match pattern* constrains the range of base models to which the aspect is applicable. It actually defines a query on the base models to which the concern specification can be woven. Finally, the *merge guidance* specifies guidelines for weaving the given aspect, or more accurately its concern specification, to any applicable base model (according to the match pattern). Note that although the same concern specification may have several pairs of suitable match pattern and merge guidance models, we consider

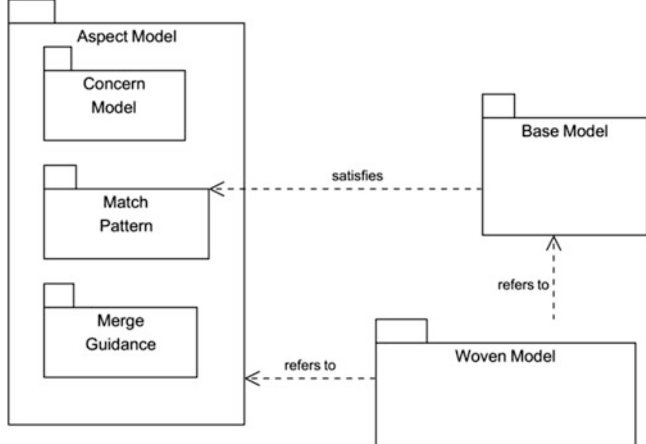


Fig. 2 A package diagram specifying the main model types and dependencies in the aspect-oriented ADOM method

an aspect model as comprised of the three aforementioned parts. In other words, several aspect models may share the same concern specification with different match patterns (and consequently different merge guidelines).

For future usage, we define an additional type of models, called *woven model*s, which are achieved after applying the rules specified in the merge guidance of an aspect model on a base model that satisfies the match pattern. Note that the resultant woven models are not required to be manually generated, especially due to their complexity. They are only used as a means for understanding the semantics of the weaving process and the complete system structure and behavior.

Figure 2 summarizes the main model types in aspect-oriented ADOM and the dependencies between them, while the rest of this section elaborates and exemplifies each type of model.

4.1 ADOM Basic Concepts and Base Models

In the context of UML, ADOM is based on a profile, depicted in Fig. 3. This profile includes five stereotypes, namely «multiplicity», «variation point», «variant», «requires», and «excludes».[2] The «multiplicity» stereotype is used for specifying the range of product elements that can be classified as the same domain, or product line, element. Two tagged values, min and max, are used for defining the lowest and uppermost boundaries of that range. For clarity purposes, four commonly

[2] ADOM actually includes a sixth stereotype, «reuse», which is out of the scope of this chapter.

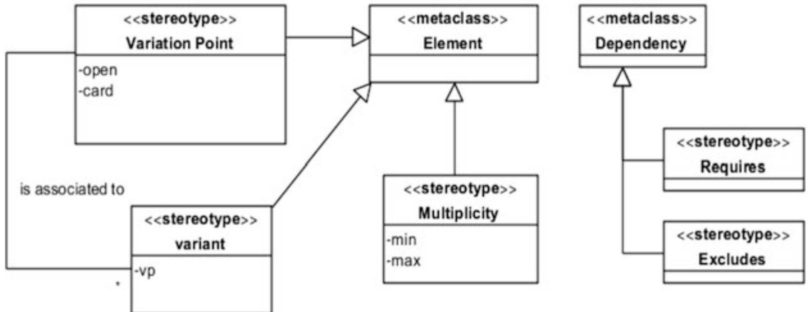


Fig. 3 The UML profile at the basis of ADOM

used multiplicity groups are defined on top of this stereotype: «optional many», where min = 0 and max = ∞, «optional single», where min = 0 and max = 1, «mandatory many», where min = 1 and max = ∞, and «mandatory single», where min = max = 1. Nevertheless, any multiplicity interval constraint can be specified using the general stereotype «multiplicity min = m1 max = m2».

Each element in the domain layer may define a variation point. This is done using the stereotype «variation point», in addition to the «multiplicity» stereotype. A «variation point» stereotype has the following tagged values: (1) open, specifying whether the variation point is open or closed, i.e., whether product-specific variants that are not specified in the core asset can be added at this point or not, and (2) card(inality), indicating the number of variant types need to be chosen for this variation point; common cardinalities are "1..1" (XOR), "1..*" (OR), "0..1" (optional XOR), and "0..*" (optional OR). Note that there are differences between the «multiplicity» stereotype and the cardinality tagged values. A variation point, for example, can be optional (e.g., «optional many») while its cardinality specification is mandatory (e.g., "1..*"), indicating that this variation point may not be included in a particular product, but if it is, then at least one of its variants (as specified in the core asset) have to be selected. Similarly, an open variation point can be mandatory (e.g., «mandatory many») while its cardinality specification is optional (e.g., "0..*"), indicating that this variation point has to be included in a particular product, but possibly use particular, product-specific variants (not specified in the core asset).

Each variant is specified using the «variant» stereotype, in addition to the «multiplicity» stereotype. A variation point and its variants should be of the same type (e.g., classes, attributes, and associations). A variant is associated with the relevant variation point via inheritance relationships. When not applicable, i.e., for variation points and variants that are not classifiers, such as attributes and operations, the relationships between variants and variation points are specified using a tagged value, vp, associated with the «variant» stereotype; vp specifies the name of the corresponding variation point. Note that the same element can be stereotyped by both «variation point» and «variant», enabling specification of hierarchies of variants.

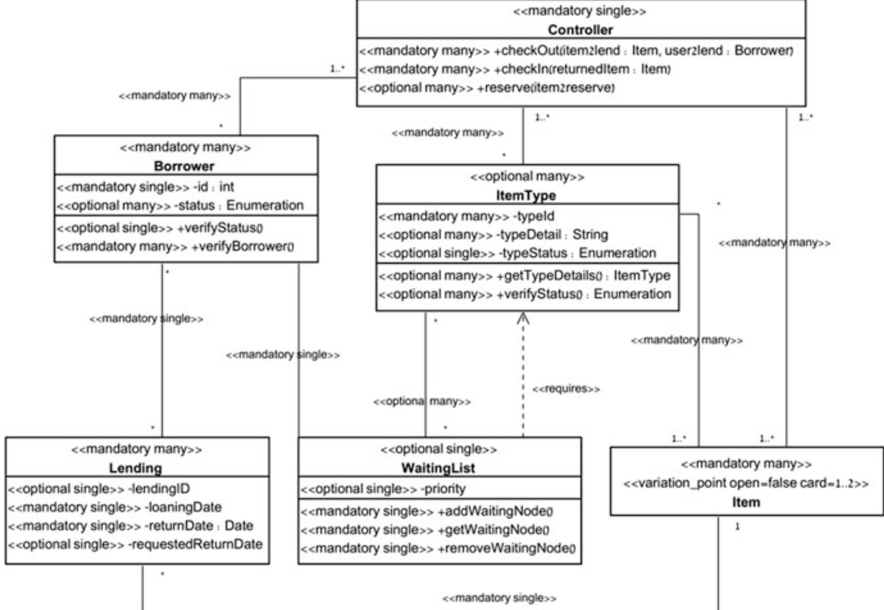


Fig. 4 A class diagram describing the structure of the CICO product line

Finally, two stereotypes are defined for determining dependencies between elements (and possibly between variation points and variants): «requires» and «excludes». A «requires» B implies that if A appears in a particular product artifact, then B should appear too. Similarly, A «excludes» B implies that if A is included in a particular product artifact, then B should not.

As an example, consider the class diagrams in Figs. 4, 5. These diagrams specify the structure of CICO applications and its possible variants. Any application in this product line, for example, must have exactly one class of controllers with different types of check-in and check-out operations and possible reserve operations; at least one type of borrowers, each of which has one attribute identifying its id, zero or more attributes denoting its status, at most one method for verifying the borrower's status, and at least one method for verifying different aspects of the borrower; and at least one class that handles the lending information, including the loaning and return dates, and optionally the requested return date and the lending id. The ability to maintain a waiting list in case the item or items are borrowed is optional, as not all the applications in the line support this functionality. However, if this ability is supported, the CICO application must include different classes of item types, as the reservation in this kind of applications is for item types rather than for individual items. This constraint is specified via the «requires» dependency between the two optional classes: *Waiting List* and *Item Type*.

Items in CICO applications must include unique identifiers and information about their overdue periods and fees. They may further have attributes specifying

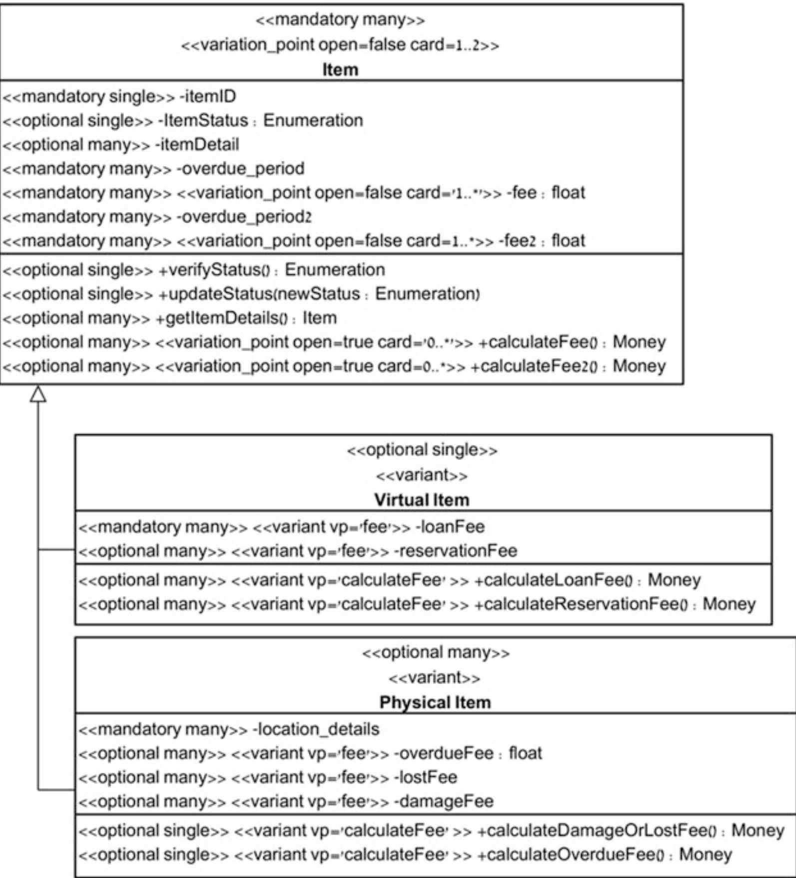


Fig. 5 A class diagram describing the item variation point

their statuses and general details. Items are primarily divided into virtual and physical items. Handling fees differ according to the item classification: based on this model, physical items may require calculating delay, damage, and lost fees, whereas virtual items may need to handle loan and reservation fees. In addition, physical items, as opposed to virtual items, have location details. According to the tagged values of the *Item* variation point (open = false and card = "1..2"), a particular product in the line cannot include items which are neither physical nor virtual. However, an item can be both virtual and physical. We could specify in the domain model an «excludes» dependency between *Physical Item* and *Virtual Item*, indicating that each product in the line may handle either physical or virtual items (but not both). In this case, the cardinality value of the variation point has to be changed to 1..1.

The sequence diagram in Fig. 6 denotes a typical check-out operation which specifies the procedure of taking or borrowing an item (and reserving it if not

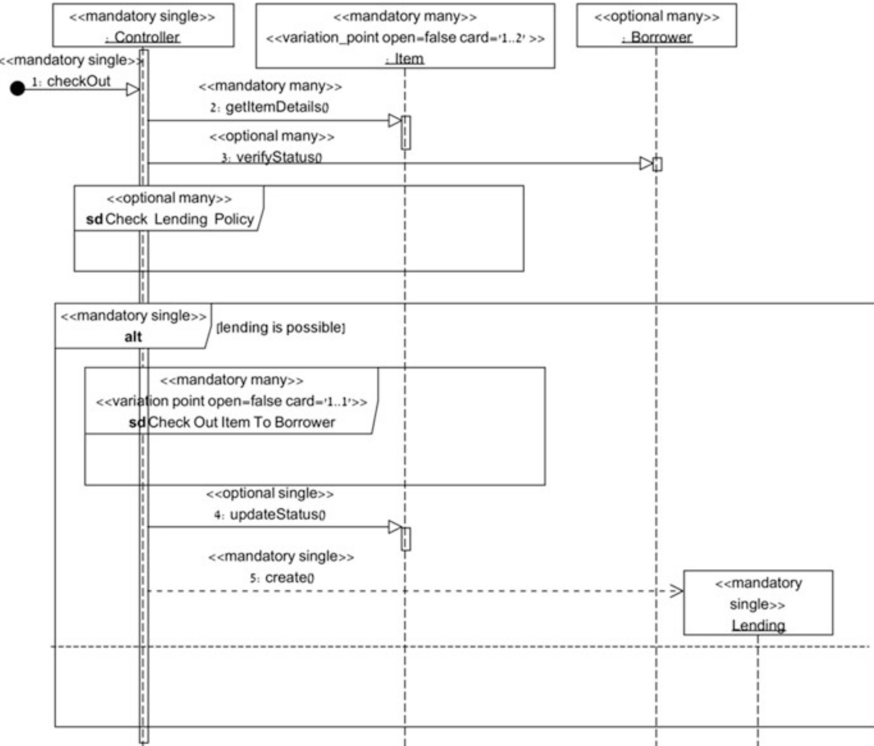


Fig. 6 A sequence diagram specifying a check-out operation

available). After invoking the check-out operation, the item details are retrieved and the borrower status may be checked. Then, the lending policy may be checked (the way this check is done is not elaborated in this sequence). If the lending is possible, the check-out operation is performed. As checking-out may differ from one application to another, this operation, which is specified as a combined fragment, is defined as a variation point. A possible variant of the operation may elaborate in a separate diagram what should be done in case waiting lists are maintained (not shown here). Finally, the item status may be updated and a lending object is created.

4.2 Concern Specification

For modeling purposes, concern specifications and base models are quite similar. However, concerns can be viewed as parts which aim to complete other modules or introduce new crosscutting functionality. Hence, they are usually smaller than base models, focus on specific issues, and do not intend to stand alone. Concern specifications (CS) can also be specified in the application or domain layers.

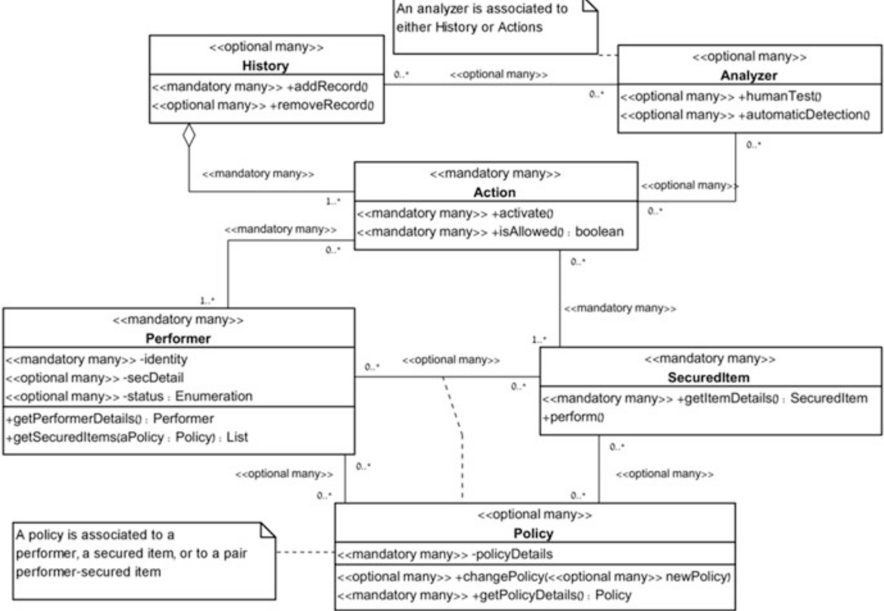


Fig. 7 A domain class diagram of the security aspect family

The model depicted in Figs. 7, 8 describes the commonality and variability allowed in a family of security aspects. Each aspect in this family must deal with performers, secured items, and actions. The class diagram in Fig. 7 captures these concepts, specifying their structure, expected behavior, and relationships. It also refers to three additional (optional) classes: *Policy* that may refer to a specific performer, a specific item, or a specific performer-item pair; *History* that may record security-related activities; and *Analyzer* that may be used, for example, for detecting different threats. Using UML sequence diagram notation, Fig. 8 describes how the allowance of a secured action is checked. This sequence can be used in different contexts, as is demonstrated later. An authorization aspect, for example, can adapt this model to reflect the requirement that resources can be used only by consumers that have been granted authority to use them.

4.3 Match Patterns

A match pattern is the part of an aspect model that specifies structural and behavioral rules and constraints on base models to which the concern specification can be woven. In other words, this part represents the minimal requirements from the base models that can weave the concern specification into them. Furthermore, as will be explained in the next section, a match pattern defines "anchors" to which the

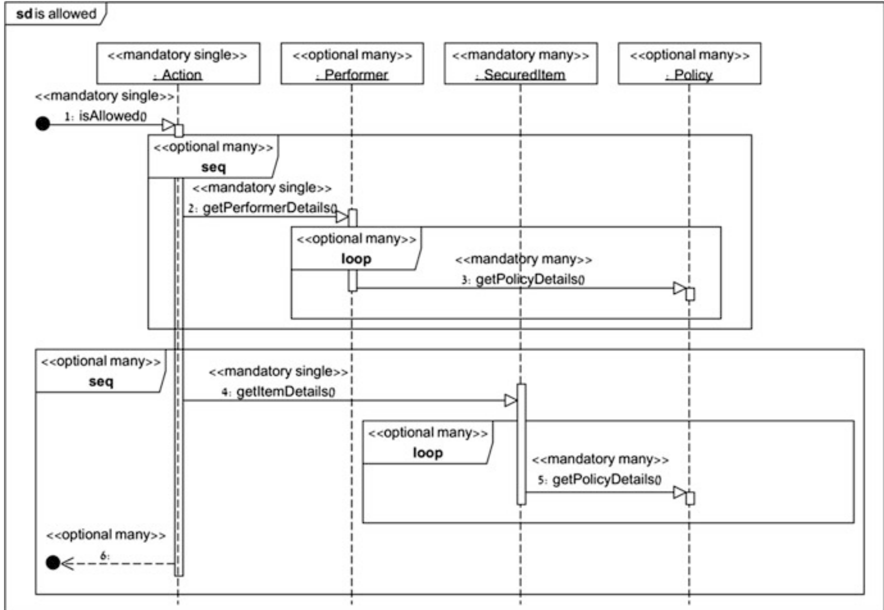


Fig. 8 A domain sequence diagram describing action activation in the security aspect family

Fig. 9 The addition to ADOM profile for specifying match patterns

merge guidance can refer. The least restricting match pattern is the empty model implying that the aspect model in general and its concern specification in particular are applicable and can be woven to any base model. Making the match pattern more detailed reduces the number of base models to which the concern specification can be woven, but enables specifying more reasonable and detailed weavings. The aim of match patterns is similar to that of join point designation diagrams (JPDD) [53]: to specify all properties that a model element must provide in order to represent a join point. However, as opposed to JPDD that defines join points on particular applications, our approach enables definitions of match patterns at the domain layer, implying the specification of similar join points to all the applications (or software products) in the product line.

For defining *matching rules*, we introduce the single stereotype profile depicted in Fig. 9. According to this profile, each element can be stereotyped as «Match-Cond», or «mc» for short. A match condition has a tagged value, named *elName*, whose value is a regular expression on the element *name*. In case the match pattern

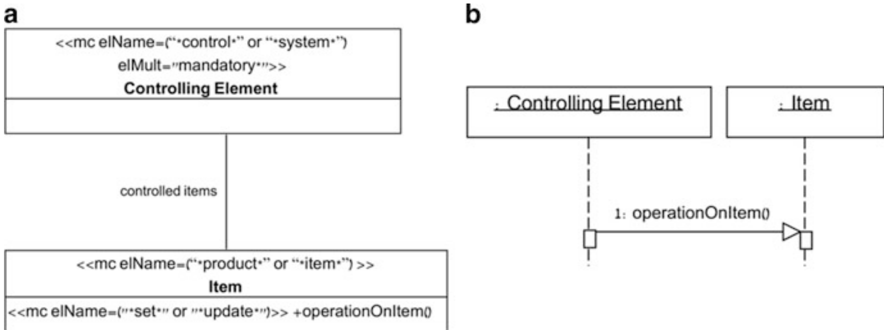


Fig. 10 The match pattern for the security aspect: (**a**) the structural constraints and (**b**) the behavioral constraints

is applied on a core asset (i.e., on models in the domain layer), it may have three additional tagged values: (1) *elMult* which is a regular expression on the element's *multiplicity* stereotype, (2) *elVP* which is a regular expression on elements that are defined as *variation points*, and (3) *elVar* which is a regular expression on elements that are defined as *variants*. At the current stage, we have not extended the tagged values of the «mc» stereotype to handle dependencies.

As an example, consider a match pattern for the family of security aspects that includes the following four rules:

1. The model includes a mandatory class whose name contains "control" or "system." This element (or these elements) will be referred to as *Controlling Element*.
2. The model includes a class whose name contains "item" or "product" and has an operation which performs some modification (i.e., the class has a "set" or an "update" operation). This element (or these elements) will be referred to as *Item* and the relevant operations—as *operation on item*.
3. The model includes an association between *Controlling Element* and *Item.* This association (or these associations) will be referred to as *controlled items*.
4. The model includes activation of *operation on item* by *Controlling Element*.

The first three rules are presented using the «mc» stereotype in the class diagram in Fig. 10a, while the forth rule is depicted in Fig. 10b utilizing sequence diagram notation. The CICO model, specified in Figs. 4, 5, and 6, satisfies the match pattern of Fig. 10. In particular, Controller from the CICO model corresponds to *Controlling Element* and Item, including its variants, namely Virtual Item and Physical Item, from the CICO model, correspond to *Item* of the match pattern, while the operation "update status" corresponds to *operation on item*. The other operations of the Item hierarchy do not match to *operation on item*, since they do not update the value of attributes. ItemType does not match Item (from the match pattern) since it does not have update operations. Finally, the association Controller-Item corresponds to *controlled items*.

4.4 Merge Guidance

The merge guidance of an aspect model combines the concern specification and the match pattern of the same aspect in order to guide the designer in how to weave the concern specification into a base model that fulfills the match pattern rules. For this purpose, the elements of the match pattern and the concern specification are used as the elements of the merge guidance.[3]

We distinguish between four types of operations: combining, concern addition, merge addition, and match only operations. A *combining operation* takes two elements, one from the concern specification and the other from the match pattern, and combines them into a third element that exhibits the features of the two combined elements. A *concern addition operation* enables the addition of elements that do not exist nor have counterparts in the base model. A *merge addition operation* enables the addition of elements that appear neither in the concern specification nor in the match pattern, but are required when merging or weaving the concern specification into a base model that satisfies the match pattern. Finally, a *match only operation* enables the specification of elements that are required only for matching base models, but are not modified as a result of weaving the concern specification into the base model.

For specifying all these merge operations the concern specification model is treated as a profile for the match pattern (i.e., all the base models that satisfy the match pattern). As an example consider the concern specification depicted in Figs. 7, 8 and the match pattern specified in Fig. 10. The elements *class 1* (Analyzer), *class 2* (History), and *class 3* (Action) in the class diagram, as well as frame (is allowed) in the sequence diagrams define concern addition operations—all these elements appear in the concern specification, but are not required in the base models by the match pattern; the associations between *Analyzer* and *Controlling Element*, between *History* and *Controlling Element*, and between *Action* and *Controlling Element* are merge addition operations—they do not appear in the concern specification nor are they required in the base models by the match pattern; they are rather required due to the merge of the aspect and the base models.

The specification of merge addition elements is similar to that of base models, using, for example, ADOM profile for specifying domain or product line artifacts; *Controlling Element* and *controlled items* define match only operations—they are the anchors for defining merge addition operations. Finally, *Item* (which is *Secured Item*) and *perform* (i.e., *operationOnItem*) define combining operations—*SecuredItem* from the concern specification is combined with *Item* from the match pattern, whereas *perform* is combined with *operationOnItem*.

All the aforementioned merge guidelines are specified in Fig. 11.

---

[3]We assume that the name spaces of the concern specification and the match pattern of the same aspect are distinctive, otherwise adding the model (package) name to the element names is required.

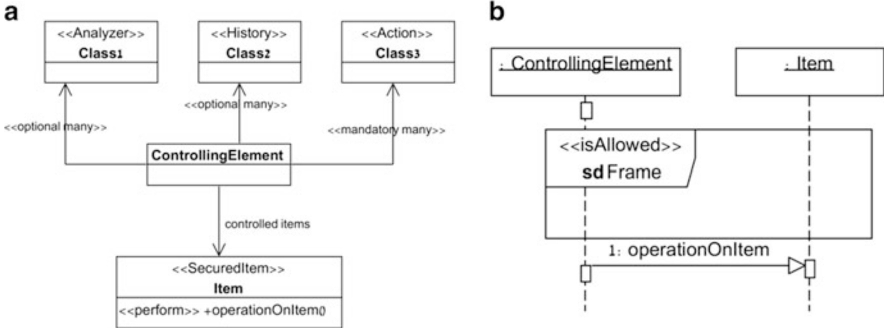


Fig. 11 The merge guidance of the security aspect to domain base models: (**a**) the structural merge and (**b**) the behavioral merge

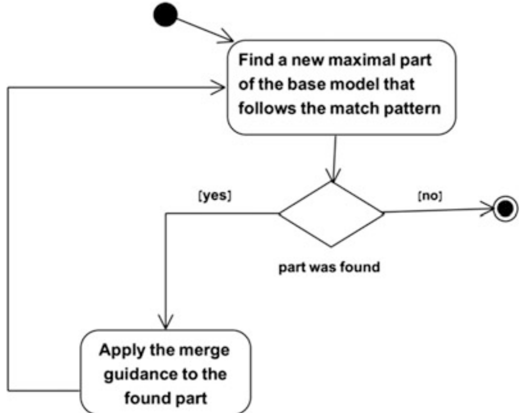


Fig. 12 An activity diagram representing the weaving process

4.5 Weaving Aspect Models into Base Models

The result of weaving an aspect model into a base model is called a *woven model*. The woven model is created by finding matches between a base model and a match pattern and applying the merge guidance for each such occurrence. Only maximal matches are used for this purpose, i.e., matches that any addition to them prevents them from being matches. Note that there may be more than one maximal match in a given base model that satisfies a single merge guidance, implying application of the same merge guidance several times to the base model (with different model portions). Figure 12 depicts the weaving process utilizing activity diagram notation.

As noted, there is a match between the CICO product line and the match pattern of the security aspect family, namely Controller from the CICO model corresponds to *Controlling Element*; Item, Virtual Item, and Physical Item from the CICO model correspond to *Item* of the match pattern, and the operation “update status” of Item

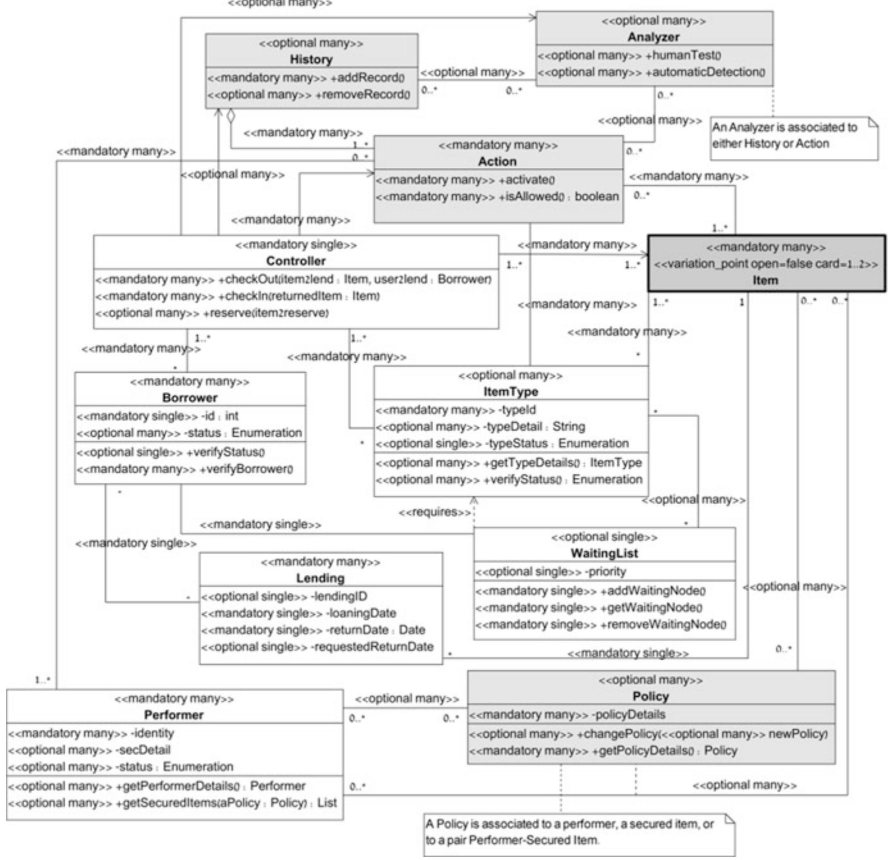


Fig. 13 The woven model resulted after weaving the security aspect into the CICO model. Part A: the structural specification modeled as a class diagram (Note that Performer was not combined with Borrower in this model, since this kind of merging was not explicitly specified in the merge guidance.)

corresponds to *operation on item*. Figures 13, 14 in the appendix present the woven model resulted after weaving the security (domain) aspect into the CICO (domain) base model. As mentioned, the woven model was generated only for the purpose of comprehending better the meanings of the aspect model parts and the weaving process. In the resultant woven model, the terminology (i.e., element names) is first taken from the base model and only afterwards (for additions) from the aspect model (i.e., from the merge guidance).

Having the woven model, either explicitly or implicitly as base and aspect models, one can use it for developing an authorized library system for a university. This system may contain two types of borrowers, `Students` and `Staff Members`; two types of item types, `Books` and `Multimedia`; two types of items, `Book`

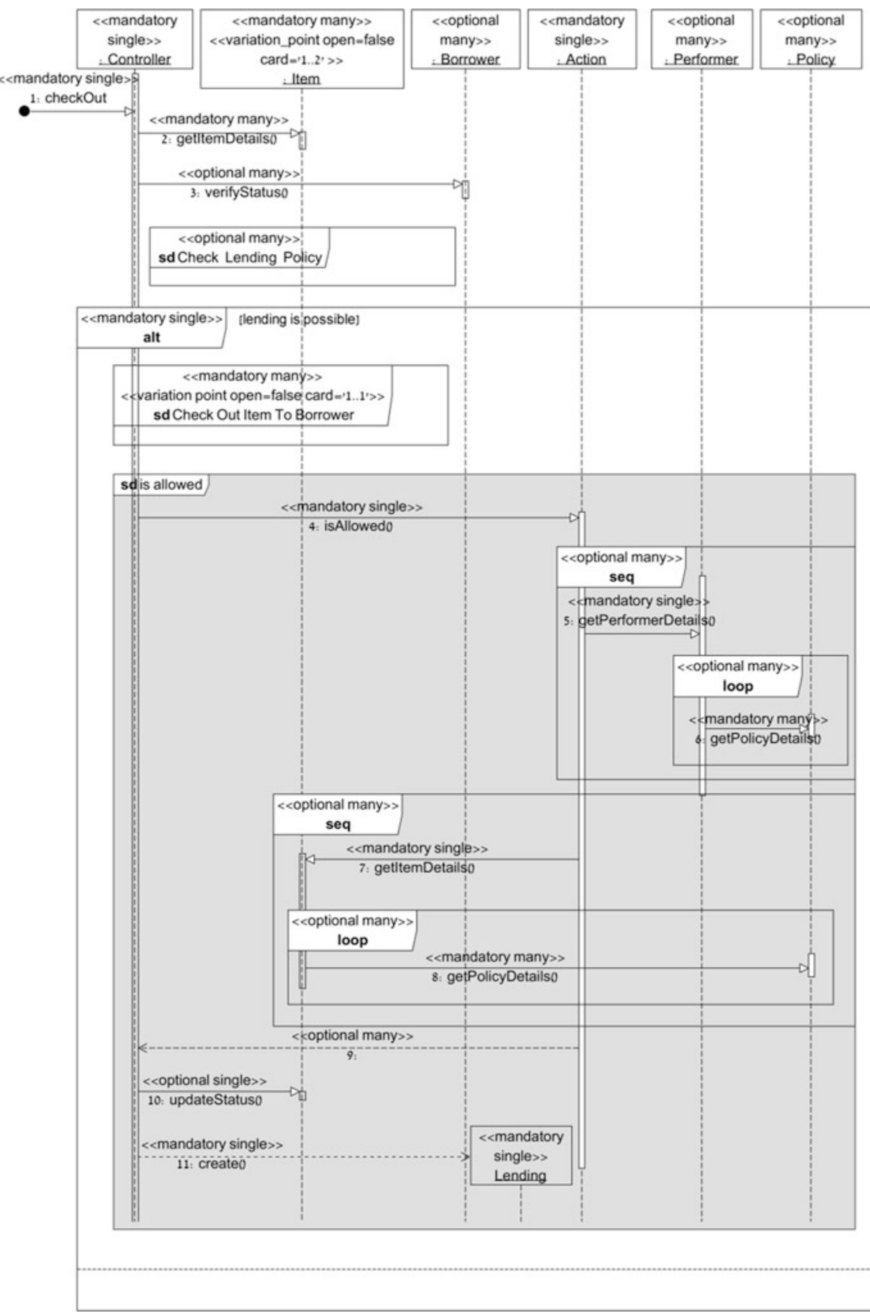


Fig. 14 The woven model resulted after weaving the security aspect into the CICO model. Part B: the behavioral specification modeled as a sequence diagram

Copies which are physical elements and Multimedia Copies which are virtual elements; one type of waiting lists, namely Book Reservations; and so on. A particular authorization aspect may enable executing actions by authorized users. For this purpose, each item, namely a Book Copy or a Multimedia Copy is connected to the allowed users, i.e., Students and/or Staff Members through Authorized Actions and Authorization Policies. No analyzer and no history recording are needed in this case.

5 Summary and Future Work

AOSD and SPLE are two approaches that aim to increase reusability of software products or artifacts. Instead of viewing these approaches as complementary or aggregative, we propose their dimensional synergy: aspect-oriented principles are used for horizontal reuse, while SPLE principles, and especially domain engineering ones, are utilized for vertical reuse of both aspects and software products. We further extend a domain engineering approach, called ADOM, to support the development of families of aspects and their weaving to families of software products. The extended method differs between three types of models, namely base, aspect, and woven models. An aspect model is further divided into three parts: (1) concern specification, which refers to issues of the aspect itself, (2) match pattern, which includes conditions on the base models to which the aspect at hand can be woven, and (3) merge guidance, which comprises guidelines and rules for weaving the aspect to any base model that satisfies the match pattern conditions. The resultant woven models define the semantics of the different models and their related operations. Each model may reside at the domain or application layer of ADOM, respectively, increasing or decreasing its level of generality.

Enabling the design and representation of families of aspects together, capturing their commonality and variability, the proposed method, aspect-oriented ADOM, increases reusability by defining weaving rules that can be applied on complete families of aspects. Furthermore, aspects can be woven to families of applications rather than to specific applications or generally to all the applications. Hence, the weaving rules can be more specific to the domain at hand and yet applied to different applications in that domain, enhancing once again the reusability of aspects. Finally, domain level aspect and base models can be used for weaving particular aspects into specific applications. This can be done by following the domain match pattern and merge guidance models and adjusting them to the specific aspect and base models.

In the future, we plan to develop a supporting tool, which will handle and manage the different model types and weaving activities. This tool, which will be part of a UML CASE tool, will automatically generate resultant woven models, check the correctness and completeness of specific base and aspect models (with respect to their base and aspect domain models), check the consistency between the different parts of an aspect model (namely, concern specification, match pattern, and merge guidance), and so on. Furthermore, we plan to specify and implement a

code generator from the suggested aspect-oriented ADOM method to different AOP languages, such as AspectJ [5]. We also consider extending the suggested approach to application engineering activities in order to support scenarios in which aspects only apply to specific features or element selection.

Appendix: The Woven Model for Secured CICO Applications

Figures 13, 14 present the woven model resulted after weaving the security (domain) aspect into the CICO (domain) base model. In these figures, the elements that belong only to the base model appear in white, the base model elements that are combined with aspect elements appear in bold and gray, and the elements that are added due to the aspect model, or more accurately due to the merge guidance, appear in gray.

References

1. Aldawud, O., Bader, A., Elrad, T.: Weaving with statecharts. Workshop on Aspect-Oriented Modeling with UML. Ensehede, The Netherlands (2002)
2. Aldawud, O., Elrad, T., Bader, A.: A UML profile for aspect oriented modeling. Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay (2001)
3. Alves de Oliveira, E., Gimenes, I.M.S., Huzita, E.H.M., Maldonado, J.C.: A variability management process for software products lines. In: The 2005 Conference of the Centre for Advanced Studies on Collaborative Research, Toranto, pp. 225–241. IBM (2005)
4. Anastasopoulos, M., Muthig, D.: An evaluation of aspect-oriented programming as a product line implementation technology. Software reuse: methods, techniques, and tools. LNCS **3107**, 141–156 (2004)
5. AspectJ. http://www.eclipse.org/aspectj/, Access date: 21/4/2013
6. Bachmann, F., Clements, P.C.: Variability in software product lines. Technical Report CMU/SEI-2005-TR-012. http://www.sei.cmu.edu/library/abstracts/reports/05tr012.cfm (2005)
7. Barth, B., Butler, G., Czarnecki, K., Eisenecker, U.: Generative programming. In: ECOOP 2001 Workshops, Panels and Posters, Budapest. Lecture Notes in Computer Science, vol. 2323, pp. 135–149. Springer, Berlin (2002)
8. Chen, L., Babar, M.A.: A systematic review of evaluation of variability management approaches in software product lines. Inf. Software Technol. **53**, 344–362 (2011)
9. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M.P., Bakker, J., Tekinerdogan, B., Jackson, A., Clarke, S.: Survey of aspect oriented analysis and design approaches. AOSD-Europe Network of Excellence. http://www.aosd-europe.net/ (2005)
10. Clarke, S.: Extending standard UML with model composition semantics. Sci. Comput. Program. **44**(1), 71–100 (2002). http://www.cs.tcd.ie/people/Siobhan.Clarke/papers/SoCP2001.pdf
11. Clauß, M.: Generic modeling using UML extensions for variability. In: OOPSLA 2001 Workshop on Domain Specific Visual Languages, pp. 11–18 (2001)
12. Clauß, M.: Modeling variability with UML. In: GCSE 2001 Young Researchers Workshop (2001)
13. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2002)

14. Coriat, M., Jourdan, J., Fabien, B.: The SPLIT method: building product lines for software-intensive systems. In: The 1st Conference on Software Product Lines: Experience and Research Directions, pp. 147–166 (2000)
15. Baniassad, E., Clements, P., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. IEEE Softw. **23**(1), 61–70 (2006)
16. Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming: introduction. Commun. ACM **44**(10), 29–32 (2001)
17. Frakes, W.B., Kang, K.: Software Reuse research: status and future. IEEE Trans. Software. Eng. **31**(7), 529–536 (2005)
18. Fuentes, L., Sànchez, P.: Designing and Weaving Aspect-Oriented Executable UML models. J. Object. Technol. **6**(7), 109–136. (2007) Special Issue: Aspect-Oriented Modeling, http://www.jot.fm/issues/issue_2007_08/article5
19. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional, Boston (2004)
20. Griss, M.L.: Implementing product-line features by composing aspects. In: Proceedings of the First Conference on Software Product Lines: Experience and Research Directions (SPLC), Kluwer, Norwell, pp. 271–288 (2000)
21. Griss, M., Favaro, J., d'Alessandro, M.: Integrating feature modeling with the RSEB. In: The 5th International Conference on Software Reuse (ICSR), pp. 76–85. IEEE Computer Society, Washington, DC (1998)
22. Groher, I., Voelter, M.: XWeave—models and aspects in concert. In: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling, pp. 35–40. ACM, New York (2007)
23. Groher, I., Baumgarth, T.: Aspect-orientation from design to code. Workshop on Aspect-Oriented Requirements Engineering and Architecture Design, Lancaster (2004)
24. Groher, I., Schulze, S.: Generating aspect code from UML models. Workshop on Aspect-Oriented Modelling with UML (held with AOSD 2003), Boston (2003)
25. Halmans, G., Pohl, K., Sikora, E.: Documenting application-specific adaptations in software product line engineering. In: The 20th International Conference on Advanced Information Systems Engineering (CAiSE'2008), LNSC, vol. 5074, pp. 109–123. Springer, Berlin (2008)
26. Halmans, G., Pohl, K.: Communicating the variability of a software-product family to customers. Softw. Syst. Model. **2**(1), 15–36 (2003)
27. Herrmann, S.: Composable designs with UFA. Workshop on Aspect-oriented Modelling, Enschede, The Netherlands (2002)
28. Kande, M.M.: A concern-oriented approach to software architecture. Computer Science PhD, Swiss Federal Institute of Technology (EPFL), Lausanne (2003)
29. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
30. Katara, M., Katz, S.: A concern architecture view for aspect-oriented software design. Software. Syst. Model. (2006) doi: 10.1007/s10270-006-0032-x, Springer
31. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), LNCS, vol. 1241, pp. 220–242. Springer, Berlin (1997)
32. Kim, D., France, R., Ghosh, S.: A UML-based language for specifying domain-specific patterns. J. Vis. Lang. Comput. **15**(3–4), 265–289 (2004)
33. Klein, J., Fleurey F., Jézéquel, J.: Weaving multiple aspects in sequence diagrams. In: Transactions on Aspect Oriented Software Development, LNCS vol. 4620, pp. 167–199. Springer, Berlin (2007)
34. Kulesza, U., Garcia, A., Lucena, C.: Generating aspect-oriented agent architectures. In: Workshop on Early Aspects, pp. 42–49 (2004)
35. Kulesza, U., Garcia, A., Bleasby, F., Lucena, C.: Instantiating and customizing product line architectures using aspects and crosscutting feature models. In: Workshop on Early Aspects OOPSLA (2005)
36. Lee, K., Kang, K.C., Kim, M., Park, S.: Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In: Proceedings of the 10th

International on Software Product Line Conference (SPLC), Baltimore, pp. 103–112. IEEE Computer Society, Washington, DC (2006)
37. Lopez-Herrejon, R., Batory, D.: From crosscutting concerns to product lines: a function composition approach. Technical Report TR-06-24, University of Texas, Austin (2006)
38. Loughran, N., Sampaio, A., Rashid, A., Bruel, J.M.: From requirements documents to feature models for aspect oriented product line implementation. In: Satellite Events at the MoDELS 2005 Conference. Heidelberg, LNCS, vol. 3844, pp. 262–271. Springer, Berlin (2006)
39. Morin, B., Barais, O., Jézéquel, J.M.: Weaving aspect configurations for managing system variability. In: 2nd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS), pp. 53–62. ICB Research Report No. 22 (2008)
40. Morisio, M., Travassos, G., Stark, M.: Extending UML to support domain analysis. In: The 15th IEEE International Conference on Automated Software Engineering, pp. 321–324. IEEE Computer Society, Washington, DC (2000)
41. OMG: Unified Modeling Language: Superstructure, Version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/ (2011). Access 21 Apr 2013
42. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)
43. Reddy, Y.R., Ghosh, S., France, R., Straw, G., Bieman, J., McEachen, N., Song, E., Georg, G.: Directives for composing aspect-oriented design class models. Transactions on aspect-oriented software development. LNCS **3880**, 75–105 (2006)
44. Reinhartz-Berger, I., Sturm, A.: Enhancing UML models: a domain analysis approach. J. Database Manag. **19**(1), 74–94 (2008)
45. Reinhartz-Berger, I., Sturm, A.: Utilizing domain models for application design and validation. Inf. Software. Technol. **51**(8), 1275–1289 (2009)
46. Reinhartz-Berger, I., Tsoury, A.: Experimenting with the comprehension of feature-oriented and UML-based core assets. In: Halpin, T. et al. (eds.) BPMDS 2011 and EMMSAD 2011, Lecture Notes in Business Information Processing (LNBIP) vol. 81, pp. 468–482 Springer, Berlin (2011)
47. Riebisch, M., Böllert, K., Streitferdt, D., Franczyk, B.: Extending the UML to model system families. In: Fifth International Conference on Integrated Design and Process Technology (IDPT), Dallas, 4–8 June 2000
48. Robak, S., Franczyk, B., Politowicz, K.: Extending the UML for modeling variability for system families. Int. J. Appl. Math. Comput. Sci. **12**(2), 285–298 (2002)
49. Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., Wimmer M., Kappel, G.: A survey on aspect-oriented modeling approaches. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.7562. Accessed 21 April 2013
50. Sinnema, M., Deelstraa, S.: Classifying variability modeling techniques. Inf. Software. Technol. **49**(7), 717–739 (2007)
51. Soffer, P., Reinhartz-Berger, I., Sturm, A.: Facilitating reuse by specialization of reference models for business process design. CAiSE'07 Workshop Proceedings, Tapir Academic Press, pp. 339–347 (2007)
52. Crawford, D.: Special issue on aspect-oriented programming. Commun. ACM **44**(10) (2001)
53. Stein, D., Hanenberg, S., Unland, R.: Expressing different conceptual models of join point selections in aspect-oriented design. In: Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD), ACM, pp. 15–26 (2006)
54. Stein, D., Hanenberg, S., Unland, R.: A UML-based aspect-oriented design notation for AspectJ. In: Proceeding of the 1st International Conference on Aspect-Oriented Software Development, ACM, pp. 106–112 (2002)
55. Stoiber, R., Meier, S., Glinz, M.: Visualizing product line domain variability by aspect-oriented modeling. In: Proceedings of the 2nd International Workshop on Requirements Engineering Visualization (REV), pp. 8–13. IEEE Computer Society, Washington, DC (2007)
56. Svahnberg, M., Van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software. Pract. Experience **35**(8), 705–754 (2005)

57. Webber, D., Gomaa, H.: Modeling variability in software product lines with variation point model. Sci. Comput. Program. **53**, 305–331 (2004)
58. Wikipedia: Security engineering. http://en.wikipedia.org/wiki/Security_engineering (2007). Accessed 21 Apr 2013
59. Yu, Y., Leite, J.C.S.d.P., Mylopoulos, J.: From goals to aspects: discovering aspects from requirements goal model. In: International Conference on Requirements Engineering, Kyoto (2004)
60. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a UML profile for software product lines. Software product-family engineering (PFE'2004). LNCS **3014**, 129–139 (2004)

Utilizing Application Frameworks: A Domain Engineering Approach

Arnon Sturm and Oded Kramer

Abstract Application frameworks aim to provide coherent code to be used and reused. The primary benefits of application frameworks stem from the modularity, reusability, extensibility, and inversion of control they provide to developers. Yet, as these frameworks become more extensive and complex, their usage becomes a burden and requires further effort. In this chapter we adopt the Application-based DOmain Modeling (ADOM), a domain engineering approach offering guidance and validation for developers when using existing knowledge, as in the case of application frameworks. The approach is adopted in the context of a programming language and demonstrated with the use of Java and is thus denoted as ADOM-JAVA. The approach preserves the regular development environment and requires minimal adaptation for using the proposed approach. We also demonstrate the use of ADOM-JAVA as a vehicle for defining and using domain-specific languages. Finally, we evaluate the use of ADOM when applied to a Java-based development. Following the guidance and validation capabilities provided by the proposed approach, the experiment shows that the productivity of the developers in terms of time and quality is expected to increase.

Keywords Application framework • Domain engineering • Reuse • Software composition

A. Sturm (✉) • O. Kramer
Department of Information Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva, Israel
e-mail: sturm@bgu.ac.il; odedkr@bgu.ac.il

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_5, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

Software development is a process that may involve the reuse of readymade and tested artifacts, such as components, executables, libraries, and frameworks. The formalization of these approaches has pervaded various application and research areas, such as software product line engineering (SPLE) [5, 24]. In SPLE there are two main phases: (1) the domain engineering phase in which the knowledge encapsulated within a domain is specified with the aim of being reused and (2) the application engineering phase in which the domain knowledge is reused and adapted as required by specific applications. To specify the reusability of the various artifacts, the software engineering community devised techniques that include generic implementation techniques for increasing software artifacts (such as models, components, and code) reuse. These techniques include generic programming that enables reuse by parameterizations, design patterns that provide solutions for specific situations, meta programming that enables programming at various levels of abstraction, as well as utilizing reflection mechanisms, and frameworks [6]. Although the design for reuse is a key issue, one should also provide mechanisms to better facilitate that reuse in an effective and productive manner. Indeed, existing tools and approaches support reuse specification at the design level; however, these mainly focus on instantiation and configuration. Moreover, when referring to programming and code, it seems that the guidance provided by the various techniques is limited and mainly provides a means for specifying reuse without proper guidance on how it should actually be performed.

In this chapter we propose an approach that aims at guiding the reuse of software frameworks (i.e., code) by adopting a domain engineering method called Application-based DOmain Modeling (ADOM) [25, 26] as an infrastructure for a new programming approach. In general, ADOM supports the reuse of many software artifacts. Nevertheless, in this chapter we apply it to a specific type of software artifact—code. We term the new approach ADOM-JAVA as we apply ADOM in the context of the Java programming language. This approach offers guidance and validation for application developers that better facilitates the domain knowledge and code reusability. The uniqueness of the proposed approach lies in the utilization of a standard programming language (including its supporting tools), and thus keeps developers within their standard development environment and the explicit reuse guidance provided within the domain knowledge.

The structure of the rest of the chapter is as follows. First, we discuss related work concerning framework usage and the reuse types. The following section presents the java agent development (JADE) framework, which is widely used for developing multi-agent systems (MAS), as a case study and a demonstrator for the problems and solutions of utilizing application frameworks. Next, we introduce ADOM—the underlining framework of the proposed approach, followed by a description of the actual application of the ADOM approach in the context of JADE and Java as the used language. To further demonstrate the use of the proposed approach, we discuss its utilization in the context of domain-specific languages. Having set the

details of applying ADOM-JAVA, we report on an initial evaluation we performed. Finally, we conclude and discuss future research directions.

2 Related Work

2.1 Software Frameworks

Software frameworks are "semi complete applications that can be specialized to produce custom applications" [9]. These frameworks are widely used in software development in general [22] and in domain-specific areas, in particular [10]. Fayad and Schmidt classified the various frameworks by "the techniques used to extend them, which range along a continuum from whitebox frameworks to blackbox frameworks" [8]. Whitebox frameworks support the reuse of their functionality by inheriting base classes and overriding predefined methods and require the developers to be familiar with their internal structure. On the other hand, blackbox frameworks support the reuse of their functionality by defining components, which meet the interface requirements and integrating these components into the specific framework and are mostly used by using object composition. Note that the whitebox frameworks are more commonly used, as developing blackbox frameworks requires much effort. Fayad and Schmidt further stress the importance of frameworks as a means for code reuse. Following their analysis, it seems that frameworks capture most of the principles provided by other approaches such as patterns, class libraries, and components. However, when constructing frameworks one should address a number of challenges, namely, *development effort*, *learning curve*, *integratability*, *maintainability*, *validation and defect removal*, *efficiency*, and *lack of standards* [8].

As determined by [22], although it is widely agreed that framework-based development improves productivity (in terms of development time and code quality), many frameworks still suffer from limited or wrong usage. This indicates that there is a need for further improvements in this kind of development. Polancic et al. [22, 23] examined the causes for this situation. They found out that the acceptance of frameworks is mainly dependent on two factors: continuous framework usage intention and the perceived usefulness of the framework. Also, their results indicate the understandability, which is "the capability of a software product to enable the user to understand whether the software is suitable and how it can be used for particular tasks and conditions of use" [14] is a major factor in using frameworks. This finding is in line with the work of Ali et al. [1], which found that applications developed by novice software developers based on provided frameworks resulted in poor software quality. This was somewhat surprising as all developers (students) seem to have abandoned the theory they were taught regarding design principles.

Summarizing the notion of framework usage, improvements are required in providing further guidance for reusing whitebox frameworks. Next, we elaborate on reuse mechanisms that can be used for guiding the aforementioned reusability.

2.2 Reuse Mechanisms

The notion of reuse has evolved over many years. Becker et al. [2] classify the reuse area into various approaches: Patterns, which define (general) templates to solve commonly occurring problems; components, which can be used and composed as is; modules, the abstract objects, which have to be instantiated to be of concrete use; and reference models (RM), which comprise information that suits various situations. In the context of this chapter, we refer to frameworks as reference models, in the sense that they fit many situations (applications). Below we describe various reuse mechanisms based on the studies of Becker et al. [2], vom Brocke [3] and Jacobson et al. [15] (among others) and present these in the context of reusing elements from the frameworks for developing applications. To demonstrate these reuse mechanisms we use a control system framework consisting of abstract concepts such as sensor, controlled element, controlled value, etc.

- *Analogy Construction*: An analogy means the transfer of information or implementation from the framework to the application, enabling the required adaptation. For example, in the case of the control system framework one can make the analogy from a sensor within the framework, to a thermometer within a climate control system or to a smoke detector which is a part of an alarm system.
- *Aggregation*: Aggregation means that one can assemble parts from the framework to construct a specific application. In the case of the control system framework, the climate control system or the alarm control system can be assembled by adopting only the relevant framework parts, such as sensors, and controlled values, neglecting other parts of the framework.
- *Configuration*: Configuration means the modification of certain framework elements following predefined rules. In the case of the control system framework, one can configure the controller to record the various measurements for data analysis or to discard this option due to performance issues.
- *Specialization*: Specialization means that application elements are derived from the framework elements by extending and/or partially modifying the more general one. In the case of the control system framework, one can specialize a sensor into remote sensor, touch sensor, etc. Thus, these can be further used within the specific application.
- *Instantiation*: Instantiation means the selection of specific values for elements within the framework for specific applications. In the case of the control system framework an object of type controlled value can be instantiated. For example in the climate control system such instantiation include the element to be controlled (e.g., room) and the thresholds for signaling.
- *Realization*: Realization means the implementation of non-implemented hooks within a framework in the specific application. This mainly refers to abstract concepts. For example, in the control system framework a controller may be an abstract object that requires implementation as different applications require different control flows.

- *Use*: Use means that one should adopt the code as is. That means that no change to the code should be done. It mainly refers to the internals of the frameworks that should not be taken care of by application developers. For example, the communication among the various components within the control system framework is hidden from the developers and should not be changed.

3 The JADE Framework

The JADE agent platform [27] is a widely used framework for developing MAS. It was developed during the last decade and is compliant with various agent standardization communities, such as FIPA, IEEE, and OMG. JADE is a complete middleware and a programming paradigm that supports the development of MAS. It consists of the services required by MAS such as communication, security, agent management, from the infrastructure point of view and a publicly available source code that enables the implementation of MAS, from the programming point of view.

In this section, we provide a general overview of the JADE framework source code via a partial model and demonstrate weaknesses in its code listing that might cause problems in using the framework. The partial JADE model in Fig. 1 demonstrates the main concepts within JADE that comprise MAS (with respect to its implementation). An agent within JADE executes various assigned behaviors that are responsible for agent functionality and agent communication with the environment (e.g., other agents). These include message passing and the scheduling and execution of multiple concurrent activities. An agent consists of many behaviors related to many types; several of these might be composite and include other behaviors. Note that each agent might be related to some ontologies and behaviors, as well as to ACL messages. Also, note that JADE code (classes) is used for two purposes: the first is to enable the proper execution of the agents and the second is to facilitate the creation of new applications through inheritance and composition. Thus, the code has two parts, one of which should not be manipulated by the application developers (although it might be stated as "public"), and the other part that should or may be changed by the developers. However, developers may not be aware of this classification, as this knowledge is partially specified within the code and guidance may also appear in other documentation. For example, usually the internal implementation part of the *Agent* class as appears in Listing 1 should not be overridden by the developers. Also, the *setup* method should be overridden as it should initialize the specific agent activities; in case the methods is not be implemented, the agent functionality will not be executed. Furthermore, it is not clear how developers should handle the *doWait* method. In the *Behaviour* class that appears in Listing 2, the methods *action* and *done* should be overridden as directed by the abstract keyword. However, note that, for example, in the *OneShotBehaviour* class (not shown here), the *done* method is already implemented and should not be handled by the developer, unless she wishes to change the semantics of the behavior. It is not yet clear how to handle the *onEnd* method. Furthermore, it is not stated

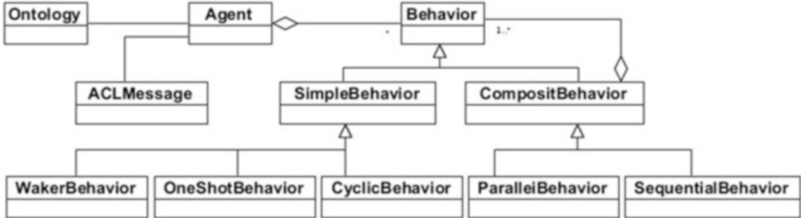


Fig. 1 A general and partial JADE model

```
public class Agent {

// internal implementation
…..
…..
    protected void setup() {}
    protected void takeDown() {}
    public void doWait() {doWait(0);}
    …
}
```

Listing 1 An excerpt of the agent class code within JADE

```
Public abstract class Behaviour implements Serializable {
// internal implementation
…..
    public abstract void action();
    public abstract boolean done();
    public int onEnd() { return 0;};
    public void onStart() {}
    …
}
```

Listing 2 An excerpt of the behaviour class code within JADE

which classes are allowed to be composed or inherited and which classes can serve as a container for other classes. Furthermore, no specification of possible control flow is provided.

Having enumerated above the missing guidelines, we call for systematic guidance to enhance reusability. For this purpose, we adopted ADOM method, which is further elaborated in the next section.

4 The ADOM Approach

ADOM [25, 26] is a method that facilitates the specification of reusable assets and their reuse within specific application regardless of the specification language. ADOM supports the representation of domain models, construction of application-specific models, and validation of the application-specific models against the

relevant domain models. ADOM is rooted in the domain engineering discipline which is concerned with building reusable assets, on the one hand, and representing and managing knowledge in specific domains, on the other.

ADOM has three layers: (1) The language layer, (2) the domain layer, and (3) the application layer. The *language layer* comprises metamodels and specifications of the languages that are used to specify the domains and application models. ADOM can be implemented in any language; however, it requires a classification mechanism. That mechanism is used for specifying constraints (and guidance) within the domain layer and for connecting application elements to their corresponding domain elements. The *domain layer* holds the reusable elements of the domain and the relations among them. It consists of specifications of various domains; these specifications capture the knowledge gained in specific domains in the form of concepts, features, and constraints that express the commonality and the variability allowed among applications in the domain, as well as guidance in how to reuse the elements within the domain. The structure and the behavior of the domain layer are modeled using the language that was defined in the language layer. The *application layer* consists of domain-specific applications, including their structure and behavior. The application layer is specified using the knowledge and constraints presented in the domain layer and the constructs specified in the language layer. An application model uses a domain model as a validation template. All the static and dynamic constraints enforced by the domain should be applied in any application of that domain.

ADOM facilitates the specification of commonality, variability, and reusability by a set of indicators which are defined as follows. The *commonality specification* in ADOM is specified by the «multiplicity» indicator, which is used for specifying the range of the number of elements within an application, i.e., the application elements, which can be classified as the same domain element. Two tagged values, min and max, are used for defining the lowest and uppermost boundaries of that range.

For *variability specification*, ADOM provides two indicators. One is the «variation_point» indicator, which has the following tagged values: (1) open, specifying whether the variation point is open or closed, i.e., whether application-specific variants that are not specified in the domain can be added at this point or not, and (2) card, which stands for "cardinality," indicating a variant's selection rule in the form of the range of variant types needed to be chosen for this variation point. The second indicator is «variant», which can be a realization of a variation point and should be of the same type. A tagged value vp associated with the variant specifies the name of the corresponding variation point.

To allow for the specification of reusability guidance, ADOM uses the «reuse» indicator, which has the following tagged values: (1) *mechanism*, which can take different values representing the different applicable mechanisms, that is, configuration, aggregation, specialization, instantiation, realization, and use ; (2) *base*, which determines whether the associated element can be used by the stated mechanism; for example, whether the element can be specialized, can compose other elements, can be configured, etc.; (3) *used,* which determines whether the element can be used

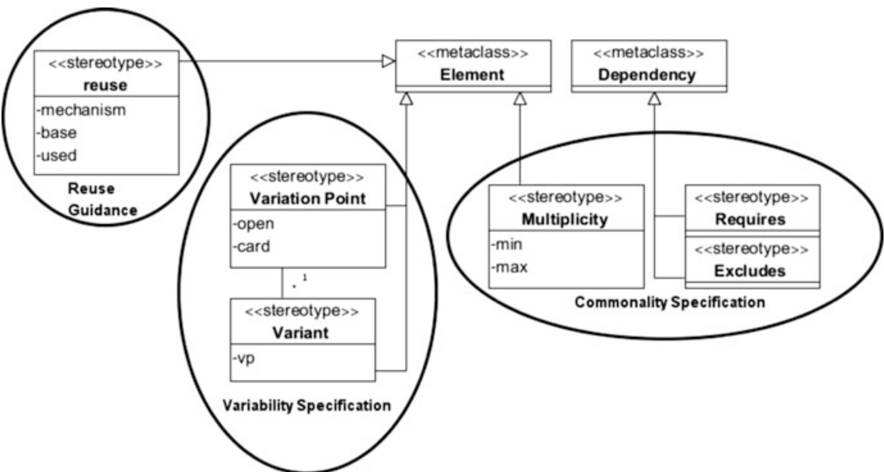


Fig. 2 A UML profile of ADOM

when applying the stated mechanism. For example, can it be composed, can it be passed as a parameter for configuration, etc.

In general, when an indicator in ADOM is associated with a domain element, its specifications are required to be fulfilled within the applications. If the indicator is not specified, then no constraints are imposed.

Summarizing ADOM indicators, Fig. 2 presents a UML profile of the various indicators—using the stereotypes classification mechanism built-in UML.

As stated before, the relations between a domain element and its specific application counterparts are maintained by a classification mechanism; each one of the elements that appear in the domain can serve as a classifier of an application element of the same type (e.g., a class that appears in a domain may serve as a classifier of classes in an application). The application elements are required to fulfill the structural and behavioral constraints introduced by their classifiers in the domain. Some optional generic elements may be omitted and not included in the application, while some new specific elements may be inserted in the specific application; these are termed application-specific elements and are not classified in the application.

ADOM also provides a validation mechanism that prevents application developers from violating domain constraints and reusability guidelines while (re)using the domain elements in the context of a particular application. This mechanism also handles application-specific elements that can be added in various places in the application in order to fulfill particular application requirements.

To exemplify the ADOM method, consider a domain with one class called *Demo* that consists of two attributes *a* and *b* and is equipped with the following indicators: multiplicity: min = 1, max = ∞; variation_point: open = true; reuse: mechanism = aggregation, base = false, used = true. That means that applications

Table 1 Demonstrating the ADOM indicators

	Multiplicity	Variation_point	Reuse
Domain specification			
Demo	1,∞	open = true	mechanism = aggregation, base = false, used = true
a	1,1		
b	0,∞		
Application specification			
App1Class(Demo) x(a)	√	√	√
App2Class(Demo) x y(b)	X	√	√
App3Class(Demo) App4Class x(a)	√	√	X

in the domain have to include at least one class classified as *Demo* and that class cannot aggregate other classes, yet, can be part of other classes. As it is also specified as an open variation point it can be extended by various application-specific variants.

Table 1 demonstrates the relationship between the domain layer and the application layer as well as the validation capabilities. Following the above example, the specification of the domain appears on the first three rows. The classification of the application elements is shown in brackets in the next three lines of the table. For the first case all constraints hold. For the second case, an attribute which is classified as "a" is missing. Since it is a mandatory attribute (see the multiplicity specification on the domain specification), it is a violation of the domain specifications. In the third case a violation occurs with respect to the reuse guidance as a "Demo" class aggregates another class—App4Class.

5 The ADOM-JAVA Dialect

In this chapter we deal with frameworks at the code level. Thus, we adopt ADOM along with Java as the underlying language. To fully apply the approach we use Java annotation[1] to satisfy the classification mechanism requirement, since it enables the specification of meta data. Listing 3 synthetically demonstrates the usage of the Java annotation in both the domain and application layers. In the domain layer the multiplicity indicator is used to constrain the domain's applications to having classes classified as *aDomainClass* at least *A* times and no more than *B* times. In the application layer the *aApplicationClass* class is classified by the *aDomainClass* class. Furthermore, the *aDomainClass must* consist of other classes and cannot be aggregated into other classes as indicated by the @reuse indicator. In addition, the *aDomainClass* cannot be specialized by the application classes and has to be used for configuration in one of the application classes. Note that no constraints exist regarding the configuration of the *aDomainClass*. In addition, in

[1]In this work we use the Java Annotation called @Java [4].

```
// domain layer code
@multiplicity(min = A, max = B)
@reuse(mechanism = aggregation, base = true, used = false)
@reuse(mechanism = specialization, base = false)
@reuse(mechanism = configuration, used = true)
public class aDomainClass{}

// application layer code
@aDomainClass
public class aApplicationClass {
    appVar bApplicationclass;
    …
}
```

Listing 3 Demonstrating the ADOM-JAVA syntax

```
//domain layer code
@multiplicity(min = 1)
@reuse(mechanism = specialization, base = true)
@reuse(mechanism = aggregation, base = true, used = false)
@reuse(mechanism = configuration, base = false, used = false)
public class Agent implements Runnable, Serializable {

@multiplicity(min = 1, max = 1)
@reuse(mechanism = specialization, base = false)
@reuse(mechanism = aggregation, base = false, used = false)
@reuse(mechanism = configuration, base = false, used = false)
private class AssociationTB {….}

@multiplicity(min = 1, max = 1)
@reuse(mechanism = specialization, base = true)
protected void setup() {}

@multiplicity(min = 1, max = 1)
public void doWait() {doWait(0);}
}
```

Listing 4 The JADE agent class with the ADOM indicators

the application layer code the *aClassApplication* class consists of the aggregation of the *bApplicationclass* via a reference variable; this is in line with the reuse specification, as it must consist of other classes. Nevertheless, even if this constraint was not specified, then the insertion of additional elements to the class is allowed and considered as application-specific elements.

As in this chapter we focus on reuse guidance, we emphasize the use of the reuse indicator and demonstrate it over the JADE code. Listing 4 presents a part of the original JADE Agent class along with the annotation suggested by ADOM-JAVA.

The annotations in the listing that appear before the class declaration have the following semantics (in the order they appear in the listing):

- There should be at least one agent class (min = 1).
- The agent class should be specialized.
- The agent class should be composed with other classes (or types) via data members but cannot participate in other containers.
- No configuration of the agent class is allowed.

```
//application layer code
@Agent
public class PingAgent extends Agent {

    private Logger myLogger =
                   Logger.getMyLogger(getClass().getName());
    private class WaitPingAndReplyBehaviour extends
                                   CyclicBehaviour {……}

        @setup
        protected void setup() {
            // Registration with the DF
            DFAgentDescription dfd = new DFAgentDescription();
            ServiceDescription sd = new ServiceDescription();
            sd.setType("PingAgent");
           sd.setName(getName());
           sd.setOwnership("TILAB");
           dfd.setName(getAID());
           dfd.addServices(sd);
           try {….}
   }
}
```

Listing 5 The use of the agent class (adopted from the PingAgent example of JADE)

Next, the annotations for the private class *AssociationTB*, which is an internal implementation of the agent that refers to its management within the JADE framework, are presented. Since the class is intended to be internal, no changes or specializations to it are allowed. The annotations for the *setup* method state that there should be only one such method and it has to be specialized (by overriding it). Finally, the annotations for the *doWait* method state that there should be only one such method; however, its uses are not dictated (as there is already an implementation of that method).

Listing 5 presents an application that extends the class Agent of JADE. Following the ADOM approach, the application elements (class, attributes, and methods) are annotated with the framework element names. This facilitates the verification and enforcements of the constraints introduced within the framework. As can be seen, the PingAgent specializes the Agent class, and there are two application-specific elements that are allowed, as stated by the @reuse(mechanism = aggregation, base = true) statement. As the *setup* method has to be overridden, it also appears in that listing. The *doWait* method did not require any changes and is reused as is. This is also acceptable, as no constraint was specified.

To this end, we have shown the way that ADOM-JAVA supports the reusability guidance related to the structural nature of the application.

Nevertheless, ADOM-JAVA is applied to the behavioral aspect as well. In the following we present this notion. Listing 6 presents the implementation of the *checkInSequence* method of the *AchieveREInitiator* class within JADE. This class implements the FIPA-Request-like interaction protocols, in which the initiator sends a single message within the scope of a specific interaction protocol in order to verify if the RE (Rational Effect) of the communicative act has been achieved or not. The structure of such a protocol is as follows. The initiator sends a message, the responder can then reply by sending a *not-understood* or a *refuse* to achieve the

```
@multiplicity (min = 1, max =1)
protected boolean checkInSequence(@multiplicity (min = 1, max =1)
ACLMessage reply) {

    @multiplicity (min = 1, max =1)
    @reuse (mechanism = use)
    String inReplyTo = reply.getInReplyTo();

    @multiplicity (min = 1, max =1)
    @reuse (mechanism = use)
    Session s = (Session) sessions.get(inReplyTo);

    @multiplicity (min = 1, max =1)
    @reuse (mechanism = aggregation, used = false)
    ifStat1: if (s != null) {
        @multiplicity (min = 1, max =1)
        @reuse (mechanism = use)
        int perf = reply.getPerformative();

    @multiplicity (min = 1, max =1)
    @reuse (mechanism = use)
    ifStat2:if (s.update(perf)) {
     // The reply is compliant to the protocol
        @multiplicity (min = 1, max =1)
        @reuse (mechanism = use)
        switchStat1: switch (s.getState()) {

            @multiplicity (min = 1)
            @reuse (mechanism = realization, base = true, used =false)
            caseStat1: case @reuse (mechanism = instantiation)
                        Session.Status:
            @multiplicity (min = 1, max = 1)
           @reuse (mechanism = use, base = true, used =false)
            caseStat2: default:

           @multiplicity (min = 1, max = 1)
           @reuse (mechanism = use)
           failure1: return false;
          }
          // If the session is completed then remove it.
        @multiplicity (min = 1, max =1)
        @reuse (mechanism = use)
        ifStat3: if (s.isCompleted()) {
             @multiplicity (min = 1, max =1)
             @reuse (mechanism = use)
              remove:sessions.remove(inReplyTo);
          }
        @multiplicity (min = 1, max =1)
        @reuse (mechanism = use)
          success: return true;
          }
       }
    @multiplicity (min = 1, max =1)
    @reuse (mechanism = use)
    failure2: return false;
}
```

Listing 6 The implementation of the checkInSequence method within the AchieveREInitiator class—a generalized version

rational effect of the communicative act, or also an *agree* message to communicate the agreement to perform the communicative act.

The implementation presented in Listing 6 is adjusted from the JADE implementation and is equipped with the ADOM-JAVA indicators. Note that in this case,

```
@checkInSequence
protected boolean checkInSequence(@reply ACLMessage reply) {
@inReplyTo String inReplyTo = reply.getInReplyTo();
@s Session s = (Session) sessions.get(inReplyTo);
@ifStat1 if (s != null) {
          @perf int perf = reply.getPerformative();
          @ifStat2 if (s.update(perf)) {
          @switchStat
          switch (s.getState()) {
        @caseStat1 case
            Session.POSITIVE_RESPONSE_RECEIVED:
        @caseStat1 case
            Session.NEGATIVE_RESPONSE_RECEIVED:
              // The reply is a response
              Vector allRsp = (Vector)
                                    getDataStore().get(ALL_RESPONSES_KEY);
              allRsp.addElement(reply);
              break;
        @caseStat1 case
           Session.RESULT_NOTIFICATION_RECEIVED:
             // The reply is a resultNotification
             Vector allNotif = (Vector)
                        getDataStore().get(ALL_RESULT_NOTIFICATIONS_KEY);
             allNotif.addElement(reply);
             break;
       @caseStat2 default:
            @failure1 return false;
          @ifStat3 if (s.isCompleted()) {
              @remove sessions.remove(inReplyTo);
                 }
         @success return true;
            }
           }
       @failure2: return false;
}
```

Listing 7 The implementation of the checkInSequence method within the AchieveREInitiator class—an implemented version

since the statements have no classifiers, we used the notion of labels to refer to these statements.

In this listing the following constraints should be imposed. There should be only one *checkInSequence* method with one parameter. The *inReply* variable should be used as is. The same holds for the *s* (of type Session). Next, the *ifStat1* should appear once in any application but may consist of additional element (i.e., statements), yet it cannot be composed into other blocks (used = false). Following the *perf*, the *ifStat2*, the *switchStat*, should appear only once and should be used as is. The *caseStat1* refers to the cases within the switch statement. In this case there might be several cases and these should be realized. Next, the other specifications of the domain elements are similar in that they should appear only once and used as is.

Listing 7 presents the application code of the checkInSequence method. The domain classification also uses Java annotation. The code can run without these annotations; yet, these are used for checking the compliance with respect to the guidelines provided within the framework code.

6 Other Applications of ADOM-JAVA

The idea of easing application development applies not only to the utilization of frameworks but to general development processes as well. Many efforts have been made in order to increase programming productivity (by reducing the development efforts) [19, 20]. For example, according to Jones [16], the development of third generation languages, which raised the level of abstraction and hide complexity, improved programming languages' productivity by 400 % compared to assembly (measured by the average number of source statements per function points). Furthermore, the object-oriented paradigm, which supports reusability through inheritance, improved programming productivity yet again. For instance, Java improved the productivity of Basic (a structural programming language) by an additional 20 % [16]. As stated in the introduction, many reuse techniques have been evolved, yet, these are general techniques and do not refer to specific domains. Current belief among the software engineering community advocates that future productivity improvements will only be achieved through utilization of commonalties in different domains [6, 7, 18, 28]. In order to assimilate this notion, domain-specific languages (DSLs) were devised [18, 21]. In general, DSLs are divided into two distinct types, external and internal DSLs, which we discuss next.

The basic premise of external DSLs is that the underlying principles of higher abstraction levels and tailoring to specific domains necessitate the development of the DSL from scratch. There is typically a domain expert with expertise in the semantics of the domain and an expert programmer with expertise in developing complicated and sophisticated software, both working on this process [18]. This process can be further divided into two sub-processes: design and implementation. The design process includes defining domain constructs and their relationships, semantics, notations, and constraints. The implementation process includes building a code generator, an optional domain-specific framework, and the DSL's integrated development environment (IDE), which consists of the DSL's supporting tools. The code generator takes the DSL specifications as input and validates them according to the domain constraints, issues error reports if necessary and finally, transforms them to low level source code as output, optionally using a domain-specific framework for this transformation. As these tools were built by experts, the resultant code reuses domain and programming expertise. The main two advantages of external DSLs are improved productivity (mainly due to abstraction) and reuse of expert knowledge. However, external DSLs still suffer from various limitations. The design and implementation of external DSLs is complicated and time consuming. Even if the work is done by experts and supporting tools are available, it might not be enough to ensure a successful working DSL. According to [12], most DSL projects are usually abandoned in the development process and the work is eventually done in regular general purpose languages. Moreover, introducing the notion of DSL-based development into an organization requires significant changes in the organization's development paradigm. These changes require both new tools and new processes.

While some managers might be able to see the long-term advantages of DSLs, others might be reluctant to introduce radical, expensive, and time-consuming changes to their natural development process. All of these reasons indicate that the applicability of external DSLs is limited. Another limitation of external DSLs is their limited expressiveness. External DSLs confine the application developer to a pre-formulated set of the language constructs. As the language is tailored to a specific domain, it cannot be used to express semantics outside of the language's boundaries, which could have been wrongfully designed.

Internal DSLs draw their inspiration from the recognized drawbacks of external DSLs. Their basic premise is that DSLs should not be developed from scratch but rather they should be embedded in existing proven general purposed programming languages (GPPLs). In this sense, internal DSLs are no different than regular domain-specific application programming interfaces (APIs). However, they are different in the sense that the APIs are designed to resemble natural languages. This is achieved by advanced coding techniques such as method chaining, expression builders, interface chaining, and generics. [11]. The main advantages of internal DSLs is that they do not suffer from the above-mentioned drawbacks of external DSLs. This improvement results from three main reasons: (1) the development of internal DSLs is much easier with respect to external DSLs, mainly because the GPPL facilities (i.e., advanced IDEs) already exist; (2) internal DSLs do not necessitate a radical change in the organization's natural development paradigm as they permit using the same set of tools (such as a programming languages, IDEs, and compilers) and processes; and (3) internal DSLs do not limit application developers' expressiveness as they allow the use of GPPL regularly. These reasons indicate that internal DSLs are more applicable than external DSLs. However, internal DSLs introduce the following limitations. External DSLs achieve improved code quality through pre-code generating validation algorithms and higher abstraction levels. Current reports of internal DSLs focus on code readability and maintainability [17]. Although these should have positive effects over productivity, it is hard to see how sophisticated APIs raise the level of abstraction similar to external DSLs. Also, although internal DSLs can exploit coding techniques in order to assure some domain semantics, they cannot implement validation algorithms that examine the specified code according to domain constraints. Ultimately, the application programmer's expressiveness is unconfined, thus she can use (or abuse) the API in any way, and therefore, internal DSLs are less productive than external DSLs.

To bridge the gaps between internal and external DSLs, we also apply ADOM-JAVA to further guide the developers, reducing their development efforts and thus increase their productivity and code quality. As we use two levels of abstraction, we gain the advantages of external DSLs and integrate the approach with GPPL we therefore gain the benefits of internal DSLs. Furthermore, the application of ADOM-JAVA may also be used for dictating architectural and programming styles.

7 Evaluation

In order to evaluate the use of ADOM-JAVA, we performed an experiment with undergraduate students to check whether the approach can lead to better code quality and to development time reduction. The research questions were the following: (1) Does the development of an application with ADOM-JAVA lead to better code quality with respect to the development of application with Java? (2) Does the development of application with ADOM-JAVA better address the functional requirements with respect to the development of application with Java? (3) Does the development of an application with ADOM-JAVA lead to a reduction of the development time with respect to the development of application with Java?

The subjects in the experiment were 50 undergraduate students in an Information Systems Engineering program at the Ben-Gurion University of the Negev, who participate in an "Object-Oriented Analysis and Design" course. During the course, the students studied ADOM and its capabilities and, in particular, the application of ADOM-JAVA. The study took place at the end of the course as a class assignment. The experiment consisted of two groups. The students in both groups received a requirement document of the application (a lab management system) and the group that uses ADOM-JAVA received the domain code as well as the supported tool. Note that the domain was familiar to all students, as it was part of the course material. The experiment lasted 9 hours in which the students had to provide a working application, which fits the requirements. The experiment took place in various sessions, where each session consists of students programming with regular Java and students programming with ADOM-JAVA. Their work was supervised to ensure that each student perform the task independently of the others. During the experiment the students were allowed to take breaks as the experiment duration was long. To encourage the students' performance, we motivated them by adding a bonus to their final grades, in accordance with their achievements.

To check the outcome of the students, we measured the development time (hours), ran a series of tests to verify the functionality (number of tests passed), and examined the code structure in terms of layer separation and object responsibility assignments. Table 2 presents the experiment results. It can be seen that the students who used ADOM-JAVA achieved better results in all categories. This was very clear with respect to the development time as the differences were statistically significant (using Wilcoxon rank sum bidirectional test) and with respect to code quality. The differences related to the functionality were a bit lower (and not statistically significant) since the number of tests was low and the application was relatively simple.

In addition to the empirical analysis we interviewed the subjects who used the ADOM-JAVA. They mentioned that the approach indeed introduces development guidance, yet further training and tool improvement is required.

Although further examination is required, following the results it can be observed that although explicit domain knowledge may be cumbersome, it does have a positive effect over the development process.

Table 2 The experiment results

		ADOM-JAVA	Regular Java	Sig.
# Participants		26	24	
Development time (hours)	Average	7.49	8.43	0.001
	Std	1.22	0.93	
# of pass tests	Average	2	1.54	0.107
	Std	1.02	0.98	
# of problems concerning layer separation		0	8	
# of problems concerning responsibility assignments		5	16	

8 Summary

Reusability has long been discussed and addressed by both practitioners and researchers. The main goal of reusability is to increase productivity in terms of reduced development time and increased code quality. Many reuse techniques have been devised, yet their usage might introduce difficulties. In this chapter we refer to one of the common reuse techniques, namely, frameworks. It is well known that frameworks provide a rich knowledge and implementation for the application that uses them. Yet, due to the amount of knowledge (and implementation) encapsulated in these frameworks, it is difficult to apply these effectively. To bridge this gap, in this chapter we adopt a domain engineering approach, ADOM, adapt it to the context of programming and frameworks, and demonstrate how that approach guides the reusability of a given framework. We also evaluate the approach and find it useful for the task at hand.

While ADOM-JAVA looks promising in increasing developers' productivity with respect to frameworks and domain-specific languages, it is clear that additional examination is required. Currently, the meta model of ADOM consists of only the reuse mechanism with two type of constraints. Thus, we plan to examine the specification of the reuse indicators to further facilitate the reuse guidance. In particular, we are interested in examining which other reuse mechanisms are required to be supported and what are the parameters needed for their configuration. In addition, the usage of ADOM-Java should be further explored and evaluated.

References

1. Ali, Z., Bolinger, J., Herold, M., Lynch, T., Ramanathan, J., Ramnath, R.: Teaching object-oriented software design within the context of software frameworks. In: Frontiers in Education Conference (FIE), pp. S3G-1–S3G-5. IEEE, Washington, DC (2011)
2. Becker, J., Janiesch, C., Pfeiffer, D.: Reuse mechanisms in situational method engineering. In: Ralyte, J., Brinkkemper, S., Henderson-Sellers, B. (eds.) Situational Method Engineering: Fundamentals and Experiences. Springer, Boston (2007)
3. vom Brocke, J.: Design principles for reference modelling—reusing information models by means of aggregation, specialisation, instantiation, and analogy. In: Fettke, L. (ed.)

Reference Modeling for Business Systems Analysis, pp. 47–75. Idea Group Publishing, Hershey (2007)
4. Cazzola, W.: @Java: A Java Annotation extension. http://cazzola.di.unimi.it/atjava.html. Last accessed April 2013
5. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional, Boston (2001)
6. Czarnecki, K., Eisenecker, U.W.: Generative Programming—Methods, Tools, and Applications. Addison-Wesley, Boston (2000)
7. Czarnecki, K.: Overview of generative software development. In: Proceedings of the European Commission and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms, France (2004)
8. Fayad, M., Schmidt, D.C.: Object-oriented application frameworks. Commun. ACM **40**(10), 32–38 (1997)
9. Fayad, M., Schmidt, D.C., Johnson, R.: Implementing Application Frameworks: Object-Oriented Frameworks at Work, 1st edn. Wiley, New York (1999)
10. Fayad, M., Johnson, R.: Domain-Specific Application Frameworks. Wiley, New York (2000)
11. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, Boston (2010)
12. Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in Java. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, pp. 855–865. ACM, New York (2006)
13. Harsu, M.: A survey on domain engineering. Report 31, Institute of Software Systems, Tampere University of Technology (2002)
14. ISO 9126.: ISO/IEC TR 9126-software engineering, product quality, quality model. International Organization for Standardization, Geneva (2001)
15. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. ACM, New York (1997)
16. Jones, C.: Estimating Software Costs. McGraw-Hill, New York (2007)
17. Kabanov, J., Raudjärv, R.: Embedded typesafe domain specific languages for Java. In: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, pp 189–197. ACM, New York (2008)
18. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, New Jersey (2008)
19. Kieburtz, R.B., McKinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D.P., Sheard, T., Smith, I., Walton, L.: A software engineering experiment in software component generation. In: Proceedings of the 18th International Conference on Software Engineering, pp. 542–552. IEEE Computer Society, Washington, DC (1996)
20. Lowell, A.J.: Programmer Productivity: Myths, Methods, and Murphology. A Guide for Managers, Analysts, and Programmers. Wiley, New York (1983)
21. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Survev. **37**(4), 316–344 (2005)
22. Polancic, G., Hericko, M., Pavlic, L.: An empirical examination of application frameworks success based on technology acceptance model. J. Syst. Softw. **83**, 574–584 (2010)
23. Polancic, G., Hericko, M., Pavlic, L.: Developers' perceptions of object-oriented frameworks—an investigation into the impact of technological and individual characteristics. Comput. Hum. Behav. **27**, 730–740 (2011)
24. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, New York (2005)
25. Reinhartz-Berger, I., Sturm, A.: Enhancing UML models: a domain analysis approach. J. Database. Manag. **19**(1), 74–94 (2008). Special issue on UML Topics
26. Reinhartz-Berger, I., Sturm, A.: Utilizing domain models for application design and validation. Info. Software. Technol. **51**(8), 1275–1289 (2009)
27. Tilab.: JADE—Java Agent DEvelopment Framework. http://jade.tilab.com/index.html. Last accessed April 2013
28. Weiss, D.M., Tau, C., Lai, R.: Software Product Line Engineering: A Family-Based Software Development Process. Addison-Wesley, Boston (1999)

Part II
Domain-Specific Language Engineering (DSLE)

Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines

Ulrich Frank

Abstract In recent years, the development of domain-specific modeling languages has gained remarkable attention. This is for good reasons. A domain-specific modeling language incorporates concepts that represent domain-level knowledge. Hence, systems analysts are not forced to reconstruct these concepts from scratch. At the same time, domain-specific modeling languages contribute to model integrity, because they include already constraints that would otherwise have to be added manually. Even though there has been a considerable amount of research on developing and using domain-specific modeling languages, there is still lack of comprehensive methods to guide the design of these languages. With respect to the complexity and risk related to developing a domain-specific modeling language, this is a serious shortfall. This chapter is aimed at a contribution to filling the gap. At first, it presents guidelines for selecting a metamodeling language. Its main focus is on supporting the process from analyzing requirements to specifying and evaluating a domain-specific modeling language.

Keywords Graphical notation of DSML • Quality of DSML • Design of DSML • Design process of DSML • Requirements analysis of DSML

1 Introduction

In recent years, the idea of using modeling languages that were designed for more specific purposes—so-called domain-specific modeling languages (DSML) —has gained increasing popularity. This is for convincing reasons: DSML promise to promote convenience and productivity of modeling, since users do not have to

U. Frank (✉)
Information Systems and Enterprise Modeling, Institute for Computer Science and Business Information Systems, University of Duisburg-Essen, Germany
e-mail: ulrich.frank@uni-due.de

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_6, © Springer-Verlag Berlin Heidelberg 2013

reconstruct technical terms on their own. At the same time, they contribute to model quality, since the concepts that are provided by a DSML should be the result of an especially thorough development process. The integrity of models, as a specific aspect of their quality, is promoted, too, since the syntax and semantics of a DSML allow for preventing nonsensical models to a certain degree. In addition to that, a DSML will often feature a special graphical notation (concrete syntax) that helps to improve clearness and comprehensibility of models.

However, designing a DSML is not a trivial task. Against this background, it is remarkable that there is hardly any method for guiding the development of modeling languages in general, the design of DSML in particular. Work on the evaluation of modeling languages, e.g., [4, 7, 12], provides language designers with criteria they should account for. However, these approaches are restricted to a few aspects only—and do not focus on guidelines for how to satisfy the suggested language features. Formal quality criteria—see for instance [13]—are more concrete, but not sufficient for guiding the design of a language because they fade out the relationship of a language to the targeted domain and to the prospective users. In recent years, method engineering, i.e., a methodical support for developing modeling methods, has gained remarkable attention. Since a modeling method consists of at least one modeling language and a corresponding process model, one would assume that work on method engineering includes support for the development of modeling languages. However, this is not the case. While there is a plethora of approaches to method engineering (for an overview see [8]), they all focus on the configuration of process models and take the modeling language as given. In a comprehensive book on domain-specific modeling, Kelly and Tolvanen describe the development and use of DSML [9]. While they provide valuable advice that is based on a number of corresponding projects, their focus is mainly on technical aspects, especially on code generation.

The current lack of support may be attributed to the fact that those who develop modeling languages are usually highly specialized experts. However, that does not mean that they are not in need for support, since the design of modeling languages can be an extremely demanding task that faces serious obstacles:

Lack of support by users: Today, a DSML is still an artifact most prospective users are not familiar with. At the beginning, they do not know sufficiently what they may expect. Often, there will be no similar DSML available that could be used as a demonstrator. The lack of knowledge about the targeted artifact will most likely not promote the prospective users' enthusiasm. But even those who understand the basic idea do not have to appreciate it: The introduction of a new language does not only imply the effort of learning it, it will also require giving up the use of GPML users are familiar with. Hence, it comes with the threat of challenging existing competence and reputation.

Amorphous requirements: As long as users lack a clear imagination of the instrument they may expect, it will be difficult to get them involved in the process of gathering and shaping requirements. On the other hand, language designers will often be not sufficiently familiar with the targeted domain to

define requirements on their own. This problem will be the more serious the less experience designers of a DSML have gained in similar projects.
Economics hard to judge: Usually, an economic justification will be most effective. That would require showing that the investment into a DSML produces a satisfactory return. However, calculating the economics of a DSML in advance is facing serious obstacles. As a consequence, it will be challenging to convince managers who decide about respective budgets.

The difficulties in analyzing the economics of DSML in advance are also related to a general design conflict. To take advantage of economies of scale, a DSML should be reusable in a wide range of application scenarios. Hence, the language concepts should not comprise too much domain-specific semantics. At the same time, however, a DSML should be a tool that provides effective support. The more domain-specific semantics a DSML includes the better its contribution to productivity—in those cases it fits.

This chapter presents guidelines for designing DSML that account for these challenges. The guidelines form the foundation of a corresponding method for developing DSML. A metamodeling method consists of a metamodeling language and a corresponding process model. Therefore, criteria for selecting a metamodeling language will be proposed. The main focus of the chapter will be on supporting the process from analyzing requirements to specifying and evaluating a DSML. The peculiarities of requirements analysis are addressed by the introduction of a structure to build and analyze *modeling scenarios*. The approach originates from work on DSML for enterprise modeling, e.g., for modeling business processes, resources, IT infrastructure, strategies, etc. The guidelines are, however, not necessarily restricted to this focus.

2 Domain-Specific Modeling Languages: Generic Requirements

While the development of a DSML will always require a thorough analysis of the targeted domain, there are also generic requirements that should be accounted for by every DSML development. Formal requirements such as correctness and completeness are certainly important, even though they may be relaxed in a particular case depending on the purpose of a DSML. Ontological criteria seem to be even more important, since they promise to guide the design of particular language concepts. Weber suggests referring to the ontology introduced by Bunge [2]. Based on Bunge's work, he proposes "ontological completeness" and "ontological clarity" as a core requirement [15, pp. 92] modeling languages should satisfy. The demand for ontological completeness is satisfied, if a modeling language covers all basic concepts proposed by the reference ontology, which was adapted from Bunge's work [14]. It includes basic concepts to describe static, (e.g., "things," "properties of things") and dynamic (e.g., "events," "states") abstractions. Ontological clarity

demands for avoiding concept overload, redundancy, or excess. A concept of a modeling language is overloaded if it maps to more than one concept of the ontology. It is redundant if there is already another language concept that maps to the same concept of the ontology. If it maps to none, it is regarded as excessive. While Weber's approach was adopted by others—e.g., [3] and [12]—it suffers from a severe misconception, which makes it especially useless for DSML. Usually, a modeling language is focusing on a particular abstraction, e.g., static, functional, or dynamic. Hence, it is not ontological complete *on purpose*—and for a good reason. Concepts offered by a DSML may be overloaded on purpose, too. Take, for instance, the concept "process" that may be part of a DSML for modeling business processes. It will usually combine static features (e.g., average costs) and dynamic features.

The following generic requirements are related to the *pragmatics* of a DSML. Hence, they account for prospective users and applications. The prospective users of a DSML include domain experts, systems analysts, and software developers. While it is likely that these groups will emphasize different specific requirements, there are three generic requirements that make sense for all users: simplicity, comprehensibility, and convenience of use. Note that these requirements are not necessarily compatible.

Requirement P1: The concepts of a modeling language should correspond to concepts prospective users are familiar with. That recommends reconstructing existing terminology. Furthermore, it recommends using graphical symbols that are suited to illustrate the corresponding concepts' meaning. *Rationale:* The more users are familiar with the concepts of a DSML and their representation, the easier it will be for them to understand and use them properly.

Requirement P2: A modeling language should provide domain-specific concepts as long as their semantics is invariant within the scope of the language's application. *Rationale:* Only if the semantics of a modeling concept is invariant within the range of its intended use, it is possible to satisfy all prospective users.

Requirement P3: The concepts of a language should allow for modeling at a level of detail that is sufficient for all foreseeable applications. To cover further possible applications, it should provide extension mechanisms. *Rationale:* A DSML is intended to cover a certain domain only. At the time of its specification, there needs to be a clear idea of this domain and related applications. However, it cannot be excluded that further aspects of the domain will be discovered that should be represented by the DSML. Extension mechanisms provide support for dealing with this change.

Requirement P4: A modeling language should provide concepts that allow for clearly distinguishing different levels of abstraction within a model. *Rationale:* Conceptual models may represent different levels of abstraction, e.g., types and—in rare cases—instances. Overloading a model with different levels of abstraction compromises an appropriate interpretation of a model.

Requirement P5: There should be a clear mapping of the language concepts to the concepts of relevant target representations. In an ideal case, all information

required by the target representations can be extracted from the model. *Rationale:* A target representation may be characterized by various semantic peculiarities, as it is, for instance, the case with object-oriented programming languages. It will contribute to risk and excessive costs, if the mapping of DSML concepts to target representation concepts is left to users.

3 Selecting a Metamodeling Language

The specification of a DSML depends chiefly on the corresponding metamodeling language (MML). The following requirements serve to guide the selection of a MML. For a more comprehensive analysis of requirements for MML, see [5].

Requirement MM 1: A metamodeling language should be supplemented by a metamodeling environment that supports the realization of model editors. *Rationale:* A DSML will usually require the use of a corresponding model editor. The development of a model editor is a major effort. Therefore, effective support from a metamodeling environment can be crucial for the economics of developing a DSML. A metamodeling environment can either support code generation from metamodels or—more preferable—allow for instantiating metamodels directly to the models used by a model editor.

Requirement MM 2: The language concepts used on different levels of abstraction, such M2 or M1, should be clearly separated. *Rationale:* If a metamodeling language does not allow for distinguishing different levels of abstraction required to specify a modeling language, this will not only cause confusion but also require substantial effort for additional constraints and tools to dissolve ambiguity.

Requirement MM 3: The graphical notation of a metamodeling language should correspond to prevalent graphical notations, e.g., of data or object modeling languages. At the same time, the notation should include elements that allow for distinguishing a metamodel from an object-level model at first sight. *Rationale:* Many language designers will be familiar with the ERM paradigm of modeling. However, representing metamodels in the same notation as models on the object level will contribute to confusion (this corresponds to the previous requirement).

Requirement MM 4: The concepts of a metamodeling language should be supplemented by a language for specifying constraints. *Rationale:* Metamodels will often leave ambiguities, which may require additional constraints.

Requirement MM 5: The concepts offered by a metamodeling language should allow for a clear mapping to concepts used for software development. *Rationale:* The efficient use of a metamodeling language requires a corresponding metamodeling tool, the development of which is facilitated by a respective mapping. In the ideal case, there is a common representation

Requirement MM 6: A metamodeling language should allow for distinguishing between different levels of abstractions. This includes especially the distinction

between characteristics of types and of corresponding instances. *Rationale:* While a modeling language is usually focused on the description of concepts, e.g., types or classes, instead of particular instances, it is sometimes required to express characteristics that apply to all instances of a type. To give an example: The concept "process" within a language for modelling business processes serves to specify characteristics of a process type. While it is a well-known fact that any process instance starts and terminates at a certain point in time, it is not possible to express this as an attribute of a process type. There are various approaches to address this requirement, such as "powertypes" [11], "clabjects" [1], or "intrinsic features" [5].

Requirement MM 7: A metamodeling language should provide concepts that allow for representing instances. *Rationale:* In rare but nevertheless important cases, it may be required to model instances. Only if a metamodeling language provides a corresponding concept, this aspect can be specified for a DSML. For instance: A DSML for modeling logistic systems may require representing particular cities.

Requirement MM 8: A metamodeling language should account for dissemination and standardization. *Rationale:* The higher the dissemination of the language, the more attractive economies of scale. Standardization contributes to protection of investment.

4 Role Model and Macro Process

Developing a DSML will often be major effort that requires division of labor. Therefore the skills and responsibilities that are needed should be accounted for in time. Especially with larger projects an explicit role model is useful to support project management. The profiles of two roles may partially overlap. A role may be assigned to one or more actors. Also, an actor may hold many roles. Typical roles include domain expert, user, business analyst, language designer, tool expert, and graphic artist. The guidelines include an extensible structure for describing roles and a corresponding set of prototypical role profiles. Table 1 illustrates the application of this structure to the role "language designer." Note that language designer, domain expert, and business analyst are supposed to play a key role in analyzing language requirements.

The specification of a DSML can be a task of remarkable complexity. It depends on intellectual creativity, especially on the ability to discover (or create) abstractions that are powerful with respect to a given purpose. The act of abstraction is a matter of individual cognitive dispositions. While the impact of a process model on cognitive dispositions is very limited, it may still help with reducing the overall complexity—thus promoting productivity and quality. Hence, the proposed process model is intended to provide an informed *orientation*—certainly not a cookbook. The introduction of a process model is usually accompanied by the advice not to interpret it as a strict sequence, but to allow for feedback loops.

Table 1 Exemplary use of structure for specifying role

Language designer	
Rationale	This role is of pivotal relevance for the development of a DSML. Language designer is not an established profession yet, nor are the required skills entirely taught in prevalent academic programs. This makes staffing a critical success factor — and may require the need to first develop the required competence
Professional training	• Should have a formal education — at best in Information Systems or Computer Science with additional studies in linguistics — that enables him to design conceptual models on different levels of abstraction • Should be trained in enterprise modeling which implies substantial knowledge about methods to analyze and design organizational action systems • Should have advanced experience in the design and evaluation of DSML and corresponding tools • Should have advanced communication skills. This includes the ability to carefully listen to people with different backgrounds and to explain the consequences of design alternatives to prospective users
Information technology skills	• Should be familiar with common approaches to specify (semi-) formal languages • Should have a background in Software Engineering. This is required to understand and overcome the semantic gap between metamodels and implementation languages
Mental capabilities	• Should have an advanced competence in developing ambitious abstractions • Should be aware of the role of language as a core instrument for structuring and managing the targeted domain • Should appreciate the use of methods (the targeted DSML would be part of) • Should be able and willing to think beyond current work practices
Attitude	• Should appreciate cross-disciplinary collaboration • Should combine a distinctive preference for language quality with an appreciation of business goals and constraints • Should show respect and empathy towards users and domain experts

For the development of a DSML, this advice is of crucial importance. The contingent nature of the subject will often require stepping back to reconsider previous assumptions. The process that is intended to guide the development of DSML is divided into a macro process and a micro process. The macro process shown in Fig. 1 is not meant as the only way to structure the overall process (a proposition that could hardly be justified anyway), but as one approach that makes sense. Subsequently, each phase of the macro process will be characterized by further details including a micro process. The phase "development of modeling tool" is included in the macro process, since it will often be necessary in order to make the DSML usable. However, the corresponding micro process is left out, because that would require to focus on software engineering aspects and thus to leave the primary focus of this chapter. Also, the selection of an MML is not accounted for in the description of the corresponding micro process because we have considered it already.

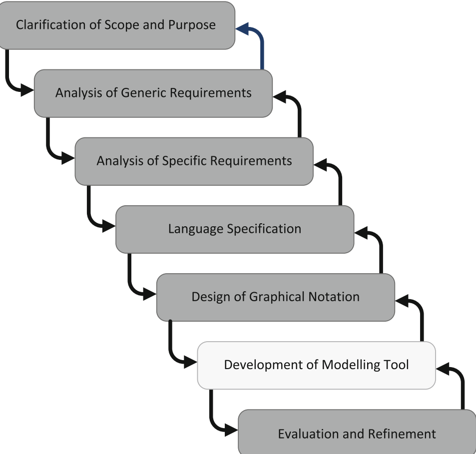


Fig. 1 Illustration of macro process

5 The Micro Processes

Each phase within the macro process is described according to a certain structure. Among other things it refers to a micro process that represents the suggested course of action for each phase of the macro process and to a set of roles, which are described in the corresponding role model.

5.1 Clarification of Scope and Purpose

The contingencies related to the conception and prospective use of DSML will usually demand for a clarification of scope and purpose. This includes the outline of a convincing motivation and rationale for the project.

Objectives: In the ideal case, the phase should produce a description of essential design objectives and a definition of the budget. However, due to the unavoidable contingencies, it may be required to first move on to subsequent phases before this outcome can be realized.

Micro Process: The rationale of a DSML will usually be related to generic prospects such as promoting productivity and quality. These need to be clarified with respect to the specific purpose and scope of the DSML to be developed. For this purpose, it is required to identify the modeling tasks that are to be addressed. These may include explicit modeling tasks of the past or other tasks, such as programming, which could be replaced by modeling tasks in a beneficial way. To get a clearer picture it is important to look at previous projects that were

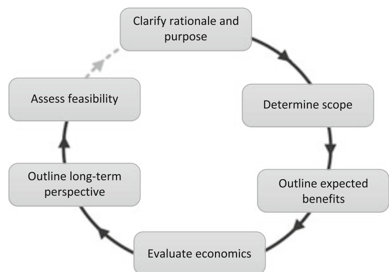


Fig. 2 Micro process "Clarification of Scope and Purpose"

aimed at these modeling tasks asking what impact a DSML would have had on performance and outcome. To outline the scope of the targeted DSML, modeling projects can be categorized with respect to modeling subject, complexity, and frequency. The higher the complexity and frequency of modeling a certain subject, the higher the benefit to be expected from a respective DSML. Also, the prospective users' attitude and skills need to be accounted for. If they are reluctant or even reject the idea of a DSML, a negative impact on economics can be expected. Against this background, one can conduct a high-level assessment of benefits with respect to certain classes of modeling tasks and corresponding users. To evaluate the economics, costs and risks of introducing, using and maintaining the DSML need to be addressed. This requires accounting for the availability of development tools, the qualification of developers and users, and changes of requirements to be expected in future times. Sometimes, the availability of a DSML will enable additional options such as the integration with other DSML or the use of models at run time. With respect to evaluating the investment into the development of a DSML, these options should be taken into account. Even if the economic perspective of a DSML seems to be promising, its feasibility may still be a challenge—especially if no previous experiences with similar projects exist. In these cases, it may be a good idea to start with a smaller scale project.

Input: Profiles provided by developers and users, outline of modeling scenarios, reports on previous modeling projects.

Participants: Manager, Business Analyst, User, Domain Expert, Language De-signer

Risks: Lack of information and knowledge may contribute to an inappropriate outline of the DSML's intended scope and to a misleading assessment of its benefits. To counter this risk, it is crucial to include respectively qualified experts.

Results: preliminary project plan, budget at least for first project phase, assignment of personnel at least to first phase, including external service providers, outline of long-term perspective

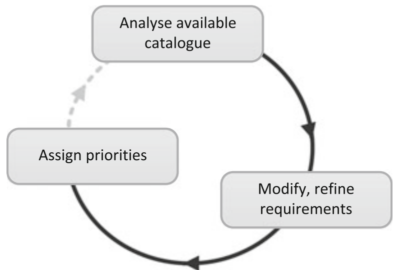


Fig. 3 Micro process "Analysis of Generic Requirements"

5.2 Analysis of Generic Requirements

The conception of DSML should account for generic requirements. They apply to every DSML, however, with different weight. Also, they may need to be adapted to a particular DSML.

Objectives: Specify generic requirements and create corresponding documentation.
Micro Process: There are not many catalogues of generic requirements available. Also, with respect to the fact that the field has not reached a mature state yet, not all proposals need to be convincing. Therefore, the analysis of available catalogues—like the one presented in Fig. 3—should pay special attention to the rationale given for each requirement.
Input: Existing catalogue(s) of and publications on generic requirements for DSML.
Participants: Domain Expert, User, Language Designer, Tool Expert
Risks: If no appropriate catalogue is available, the development of generic requirements is a cumbersome activity that implies the risk to miss requirements. Even if one can build on an existing catalogue, there is no guarantee that it is comprehensive.
Results: Catalogue of generic requirements. Each requirement should be described and justified with respect to the purpose of the DSML. Also, each requirement should be characterized with respect to its relevance.

5.3 Analysis of Specific Requirements

As already elucidated above, it is probably the most challenging peculiarity of developing a DSML that specific requirements can often not be analyzed in a straightforward way. Therefore, there is need for an approach that does not rely on merely analyzing existing modeling tasks and asking users for their expectations.

Objectives: Develop a comprehensive list of specific requirements.

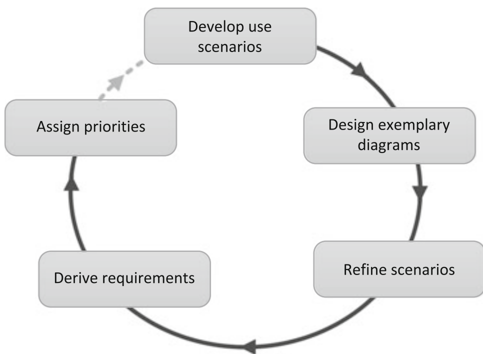


Fig. 4 Micro process "Analysis of Specific Requirements"

Micro Process: To analyze specific requirements, users, business analysts, and language designers need to be supported with developing a clear idea of what they may expect from the DSML. Our experience with developing DSML suggests that one approach is especially suited for this purpose. Based on use scenarios that are developed with respect to previous and future tasks, the potential use of the DSML is illustrated through the design of preliminary diagrams. These serve as a medium for further refining the use scenarios. To support the derivation of specific requirements, it has proven successful for us to describe the scenarios in a certain structure.

For developing use scenarios relevant modeling scenarios from the past should be identified and described. In addition to that further possible use scenarios may be developed. This can be promoted by presenting rudimentary scenarios which are then further refined. Each scenario is related to a certain diagram type. To get an idea what information should be represented in respective diagrams, one can start with a rudimentary graphical representation and then develop a list of questions that are related to the diagram. With respect to preparing for a corresponding modeling tool, it is helpful to specify for each question whether it can be answered by a machine (**A**), by a human only (**H**), or in a partially automated way (**P**). Note that the stages "develop use scenarios," "design exemplary diagrams," and "refine scenarios" are interweaved. Each diagram type should be clearly described with respect to its purpose and its key concepts. Furthermore, it should be related to other diagram types that might supplement it with respect to specific purposes. The actual example diagrams have an important function especially for novice users, since they provide an illustration of what they might expect from the intended DSML. Therefore, designing exemplary diagrams should account for an illustrative graphical representation—even though it will be replaced by a more professional notation later on. The notation used in the exemplary diagrams should be discussed and possibly further developed with prospective users to gather hints for the design of the final graphical notation. Based on the example diagram and the extensible list of questions, all participants—supported by domain experts and language designers—are supposed to commonly develop specific requirements. If a

requirement poses a substantial challenge to language design or the realization of a corresponding tool, it should be marked as such. They need to be addressed at the design stage at the latest and may result in relaxing corresponding requirements.

The following example illustrates this approach. It shows a use scenario for a prospective DSML to model services, which serves the purpose to support documenting, analyzing, and (re-) designing a system of services. A corresponding example diagram would be designed through multiple refinements during the course of asking questions such as:

- What is the share of services that are obtained from external suppliers? **A**
- How is the average level of service satisfaction? **A**
- Are there similar services that could be gainfully combined? **P**
- Which business processes are supported by a service? **A**
- Which services are potential subjects of outsourcing? **H**
- Are there service contracts that need improvement? **P**

Figure 5 shows a corresponding example diagram that serves to produce a common understanding of the range of tasks to be supported by the DSML. It also contributes to identifying specific requirements and particular design challenges—which are illustrated in the following examples.

Specific Requirement SR1: It should be possible to specify different types of services.
Specific Requirement SR2: There should be concepts that allow for defining associations between services and between services and other relevant concepts such as business processes, software systems, and organizational units.
Specific Requirement SR3: The language should provide concepts for specifying service contracts on various levels of detail.
Challenge C1: In order to promote reuse, i.e., modeling efficiency, and model integrity, it would be good to provide concepts that allow for expressing commonalities of a set of service types. For this purpose, specialization relationships would be particularly useful. However, the specification of a specialization relationship implies a remarkable challenge: In an ideal case, specialization does not only mean that a specialized concept includes the features of its superordinate concept. Furthermore it implies that an instance of the specialized concept can substitute an instance of the superordinate concept. A possible solution could be to relax the ideal concept of specialization—e.g., to give up the demand for substitutability.
Input: Previous modeling scenarios; collections of complex analysis and decision tasks, especially those that require accounting for multiple perspectives.
Participants: Business Analyst, Domain Expert, Language Designer, Manager, Tool Expert, User; optional: Graphic Artist.
Risks: If all participants lack a background in modeling with DSML, it may be difficult or even frustrating to develop appropriate scenarios and corresponding diagrams. This, however, is crucial for the analysis of requirements. While the scenarios and corresponding exemplary diagrams are an important instrument

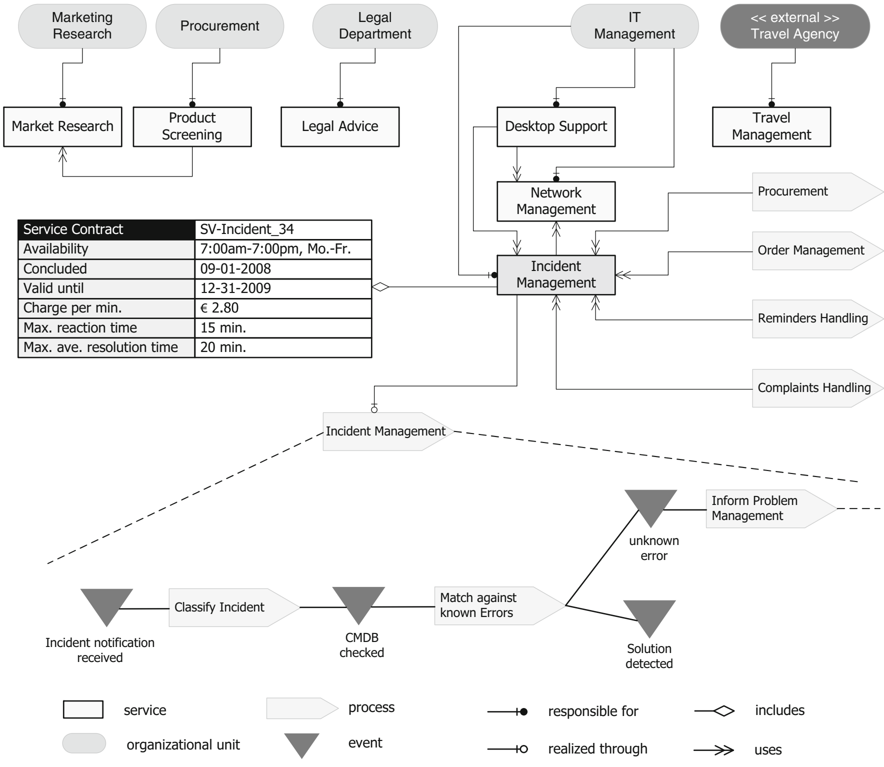


Fig. 5 Example diagram to illustrate a possible use scenario

to analyze requirements by illustrating the purpose of the prospective DSML, they may also compromise the analysis of requirements by restricting the scope to particular aspects. Therefore, selecting scenarios and creating exemplary diagrams require experienced domain experts and language designers that have an idea of how the targeted DSML could look like, but are open minded enough to appreciate suggestions and requests made by other participants.
Results: Documentation of specific requirements and corresponding rationale; documentation of specific challenges.

5.4 Language Specification

The specification of the abstract syntax and semantics is the pivotal part of designing a DSML. It requires accounting for the range of potential applications. It will usually include various design decisions, some of which are common in conceptual modeling while others are specific for the design of metamodels.

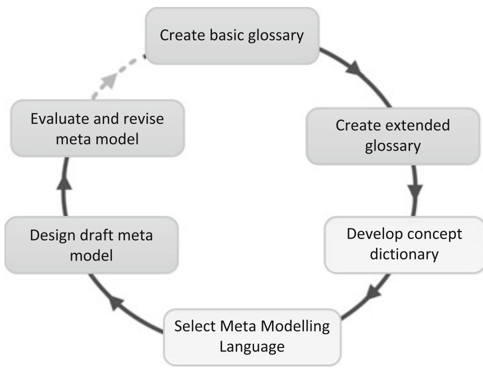


Fig. 6 Micro process: "Language Specification"

Objectives: Specification and documentation of metamodel and corresponding constraints.

Micro Process: The specification of a DSML is directed towards reconstructing concepts of the respective domain of discourse. For this reason, it makes sense to first develop a glossary with key terms. At its first development step that we refer to as "basic," the glossary is a dictionary of terms with corresponding descriptions. The basic glossary serves as a collection of terms that are used in the targeted domain of discourse. That does not mean, however, that each of these terms is suited to be included in the DSML. Instead, it needs to be decided whether a term should be reconstructed as part of the intended DSML or whether it should rather be specified with the DSML. To support this decision, we use an extended structure of the glossary that reflects key decision criteria. It is documented in [6]. The conception of DSML that we favor is aimed at a covering a wider range of (re-)use. In other words: Usually we do not develop a DSML for just one project as it is suggested e.g., by [9]. Therefore, it is important that the semantics of a language concept is *invariant* throughout the entire domain and in time (*Requirement P2*). To promote comprehensibility and usability of a DSML it should be avoided to include concepts that are not needed. Hence, each collected term should be checked for its *relevance*. A language concept is—usually—intended as an abstraction over types. These considerations result in two demands: First, the semantics of the respective instances should vary—otherwise it would literally not make much sense to bother with instantiations. Second, it should be checked whether possible instances are perceived intuitively as types. Table 2 shows an exemplary entry of a concept dictionary. The suitability of a concept is judged by evaluating the above criteria, where "+" indicates a good fit, "–" indicates that a concept is inappropriate with respect to a certain criterion, and "c" is used to express that it is contingent.

Subsequent to assessing the collected terms, it is required to decide for each term whether or not to include it in the DSML. This decision recommends reflecting upon the intended scope, i.e., the targeted domain of the language. As a consequence, one may decide to exclude a term from the language or to narrow the scope to an

Table 2 Exemplary excerpt of concept dictionary

Organisational unit		
An organisational unit is a part of an organisation that reflects a permanent principle of the division of labour. An organisational unit may contain other organisational units		
Example instantiations	"Division Electronic Devices", "Marketing Department", "Car Manufacturing Plant", "Human Resources Department"	
(a) invariant semantics	The term is used on a high level of abstraction. The essence of its semantics should not vary substantially	+
(b) relevance	This is a key term for describing organisation structures	+
(c) variance of type semantics	The semantics of types of organisational units can vary significantly	+
(d) instance as type intuitive	At least some of the potential instances will be regarded as types almost intuitively, e.g., "Department", "Division". Other potential instances, such as 'Marketing Department', "Consumer Electronics Division", will probably not be regarded as types by many. Hence, the final assessment of this criterion depends on the recommended instantiation	c

extent where a particular term is characterized by invariant semantics. This step results in a revised version of the extended glossary. Finally, entries in the glossary can be supplemented with respective designators in further languages to support international use. Before specifying the language with a metamodel, a concept dictionary can be created. It serves two purposes: On the one hand, it should prepare for the construction of the metamodel; on the other hand, it serves as a documentation of the terms specified in the metamodel. The structure of a concept dictionary reflects specific design decisions to be made with the construction of metamodels. For this purpose, it does not only represent the collection of attributes and associations that are required for the specification of a concept. In addition to that it proposes a structure to support a more elaborate description. Attributes are separated into two groups. Attributes on the type level are supposed to describe characteristic features of a type that are instantiated from the respective meta type—independent from its own instances. This is different with the category "Attributes with references to instance level." It serves to group those attributes of a type that can be initialized only by accounting for its instances. For example: an attribute such as "averageNumberOfInstances" would require counting the number of instances of a process type within a certain time period. A concept dictionary can grow to a remarkable size. Creating—and maintaining—can therefore require a substantial effort. It is a particular challenge to synchronize it with corresponding parts of the glossary and the metamodel. Therefore, creating a concept dictionary will usually make sense only if there is a tool that synchronizes corresponding parts of the dictionary and the metamodel. The next step concerns the selection of a metamodeling language and a corresponding metamodeling tool (see Sect. 3). It is optional: Often, there will be no choice anymore because the use of a certain metamodeling language is mandatory.

Like with any other complex abstraction, the design of a metamodel will usually require demanding decisions. Often, they relate to conflicting design goals. All design decisions that do not seem to be trivial should be documented using a common structure, e.g., "problem description," "design alternatives," "selected alternative," "rationale." In addition to design decisions, the specification of a metamodel is also based on principle attitudes and styles of the language designer. For instance, some designers prefer to specify as much of a DSML's semantics through the abstract syntax as possible, while others tend to keep the abstract syntax simple and represent semantics with additional constraints. Both styles have specific advantages and shortcomings. By putting more emphasis on additional constraints, a metamodel can be kept simpler. Also, representing semantic constraints on a syntactical level will often require introducing artificial concepts. Therefore, aiming at a lean metamodel with additional constraints may—at first sight—contribute to improved comprehensibility. However, for many observers, a larger number of formal constraints will not improve the readability of a metamodel. With respect to implementation, this style delegates more responsibility to programmers. Usually, the meta types of a metamodel will be represented as classes in a corresponding model editor. In the case of emphasizing syntactical specification, the correctness of larger parts of a model can be checked on the class level. In the other case, constraints have to be implemented and checked against particular instance states. Therefore, with respect to integrity, there is good reason to avoid specifying "types" through instance states. However, at the same time, introducing additional meta types to emphasize syntactical specification may threaten integrity: Certain changes applied to a model are likely to require class migration—e.g., migrating an instance representing an attribute from the class "SimpleAttribute" to the class "Attribute." Class migration is not only costly, but also risky. Due to conflicting requirements, choosing a particular specification style is also a matter of subjective preferences. To guide the reader of a metamodel with a better understanding, it is a good idea to briefly comment on the preferred specification style. The examples in Fig. 7 illustrate the difference between specification styles that put more emphasis on semantics or on syntax, respectively.

As soon as the specification of a metamodel is completed, an intensive and thorough evaluation is mandatory. This is mainly for two reasons: First, a metamodel usually includes specific pitfalls related to subtle differences in abstraction levels, neglected modeling scenarios or inappropriate constraints. Second, a language specification should be as stable as possible, because later changes will usually result in costly adaptations of models and especially of related tools. Unfortunately, the evaluation of a metamodel faces particular challenges. Even more than other conceptual models, a metamodel cannot be tested directly against targeted domain languages. This is for various reasons. In general, the concepts specified through a metamodel are linguistic constructions that include a prescriptive element in the sense that they propose how to structure the targeted domain. Therefore, the concepts may intentionally differ from existing technical concepts, because they are supposed to be superior with respect to certain purposes. However, at the same time, they should correspond to terms prospective users are familiar with, in order to foster

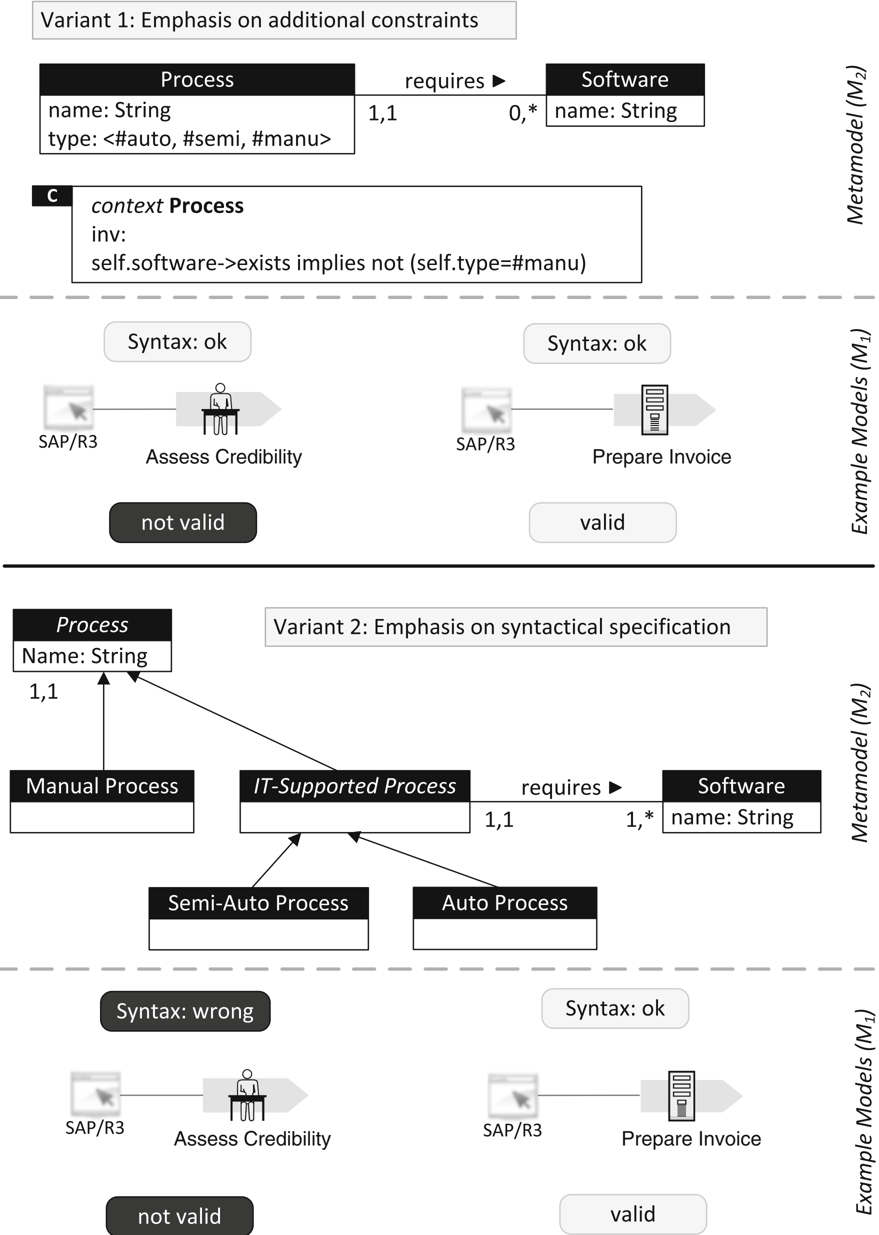


Fig. 7 Illustration of prototypical specification styles

comprehensibility and usability (*Requirement P1*). Conflicting goals like this are a further reason why a simple comparison against an existing terminology will often not be sufficient. It is certainly a good idea to involve prospective users. However, it

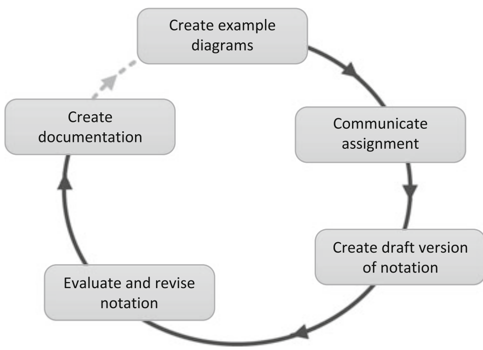


Fig. 8 Micro process "Design of Graphical Notation"

may be asked too much of prospective users to judge a metamodel directly. Instead, the concepts could be illustrated by showing how to use them for exemplary modeling purposes. The revised metamodel needs to be documented comprehensibly.

Input: Examples of metamodels, previously developed conceptual models of the targeted domain, guidelines for designing metamodels.
Participants: Language Designer, Domain Expert, User, optional: Tool Expert.
Risks: The specification of a DSML will usually face remarkable challenges. Among other things, they relate to conflicting requirements and to the differentiation of language and language application (see above). Language designers with too little experience may overlook these challenges and produce metamodels that are not satisfactory. On the other hand, language designers who are aware of the specific problems may struggle for a long time without developing convincing solutions.
Results: Metamodel, preferable specified with a metamodeling tool that allows for further transformation; extensive documentation

5.5 Design of Graphical Notation

It seems reasonable to assume that the graphical notation is of considerable relevance for the comprehensibility, usability, and productivity of a DSML. This assumption is backed by various studies (for an overview, see [10, p. 758]). Therefore, it will usually be no good idea to regard the graphical notation as a meaningless (in the sense of formal semantics) and, hence, marginal feature of a modeling language. Instead, it is recommended to design it with special care and consideration.

Objectives: Design and document graphical notation.
Micro Process: The creation of a graphical notation is a special challenge for at least two reasons. First, there is not much solid ground—in the sense of a theoretical foundation—to build on. Moody proposes probably the most

ambitious approach to address this problem [10]. However, as we shall see it is still not sufficient to clearly guide the design of a graphical notation. Second, those who are experts in the specification of a modeling language focus on semantics and abstract syntax. Usually, they are no experts in the design of a graphical notation. Sometimes, they will not even be interested in this aspect of language design. Third, those who are trained in the design of iconographic symbols will often lack knowledge about the use of DSML. Even though the existing theoretical background is not satisfactory yet, there are a few guidelines that are backed by theoretical considerations. Therefore, we suggest accounting for existing guidelines—but not without carefully analyzing how to apply them appropriately. The following guidelines reflect our own experience and also build on suggestions in the respective literature.

Guideline 1: Build semantic categories of concepts. A category should be characterized by clear features and be intuitively distinguishable from other categories. *Rationale:* Semantic categories are an important prerequisite for designing expressive and discriminating symbols. Example: In process modeling one key category could comprise processes and a further category could comprise events.

Guideline 2: Create generic symbols for each category. On the one hand, the concepts covered by a category should be represented by variations of the respective generic symbol. On the other hand, generic symbols of different categories should be distinguishable at first sight. *Rationale:* This guideline should foster the appropriate perception and interpretation, hence, the comprehensibility of a graphical notation. It is further refined in guideline 3.

Guideline 3: The bigger the semantic difference between two concepts, the bigger the graphical difference of the corresponding symbols should be. Moody speaks of "visual distance" [10, p. 763]. This principle applies both to the semantic differences between categories and to the semantic differences between concepts of a particular category. While there is lack of a convincing precise definition of visual distance, notational differences can be created through shape, color, or text (see guideline 5). *Rationale:* Using a modeling language effectively requires discriminating quickly between concepts. The more different two concepts look, the faster discrimination should be possible. At the same time, a low visual distance should help with identifying and appropriately using similar concepts.

Guideline 4: Prefer icons over shapes. This principle supplements the previous one. While two geometric shapes can be clearly different, e.g., a circle and a rectangle, they lack a reference to the represented concept. An icon is characterized by creating such a reference. This can be realized by an iconic representation of an object, e.g., a letter or a computer, or by the representation of a certain characteristic feature, e.g., symbol depicting a lightning for representing an exception. *Rationale:* Including a perceptual reference into a symbol will contribute to a more intuitive understanding of a notation and, hence, improve its usability and its suitability as a communication medium (see also *Requirement P1*).

Guideline 5: Combine shape (including icons), color, and text effectively. There are studies in cognitive psychology which suggest that shape is a more effective instrument to accomplish visual discrimination [10, p. 763]. Nevertheless, Moody's resulting recommendation is not entirely convincing: "For this reason, shape should be used as the primary visual variable for distinguishing between different semantic constructs" (ibid). Instead, a more differentiated approach seems to be better suited. In general, semantic categories should be represented by shapes (or icons). This will usually involve the use of colors. The same principle applies to concepts of a certain category, too. However, there are exceptions to this rule. On the one hand, color can be used as an additional discriminator. This makes sense, if a concept is very similar to others so that it would be difficult and/or misleading to define a separate symbol for representing it. In addition to that, color can be used as an instrument for users to express additional meaning that is not part of the DSML. Text can be used similarly to color. It is of particular importance if the variety of concept occurrences is too large to be covered by symbols. For example: Expressing multiplicities can be accomplished through a set of symbols as it is done in various flavors of the ERM. Apart from the question how comprehensible these symbols are for people who do not use them on a regular base, they are restricted to a given set of multiplicities. If the concept of multiplicities should allow for defining any specific upper and lower bound (as long as the upper bound is greater zero and greater or equal to the lower bound), such an approach simply does not work anymore. Text as an instantiation of a modeling concept is inevitable whenever it is required to name concepts of a model. *Rationale:* Each representation type has specific advantages that need to be evaluated against the requirements a DSML is supposed to satisfy. Therefore, combining representation types appropriately allows for improving a DSML's usability.
Guideline 6: Avoid "symbol overload" [10, p. 763]. A symbol should be clearly assigned to one particular modeling concept. *Rationale:* Overloading symbols would contribute to ambiguity and confusion.
Guideline 7: Avoid redundant symbols [10, p. 762]. *Rationale:* If a concept can be expressed by alternative symbols, both modellers and model observers are stressed with additional cognitive effort without additional benefit.
Guideline 8: Represent monotonic semantic features of a concept through compositions of symbols. Sometimes, concepts of a DSML need to be further refined to allow for expressing a more specific meaning. If this is accomplished by adding features in a monotonic fashion, each of these features can be represented by a respective symbol. A particular occurrence of the concept will then be represented by a composition of related symbols. *Rationale:* Combining graphical elements contributes to a more systematic construction of a graphical notation that is in line with the semantics of the related concepts. It also helps avoiding overwhelming users with huge palettes of prefabricated symbols. With respect to building tools, it implies a larger effort for implementing the composition of more complex graphical symbols. At the same time, it improves flexibility.

In addition to principles which guide the design of graphical symbols that represent certain modeling concepts, further notational elements may be required to cope with the complexity created by large models.

Guideline 9: A graphical notation should include symbols that allow for reducing diagram complexity. On the one hand, these are symbols that represent compositions of a set of other symbols. If required, they serve as a starting point for decompositions. A typical example would be a symbol used to represent aggregate processes. On the other hand, special symbols can be introduced that allow for depicting a set of identical diagram parts by one common representation. *Rationale:* Lowering the visual complexity of a diagram can substantially contribute to a better understanding and, hence, to increased productivity.

In order to apply the suggested guidelines appropriately, it is mandatory to develop a clear idea of the prospective language users and the modes of use to be covered. This recommends answering the following questions:

- Are prospective users already familiar with graphical modeling languages?
- Are they supposed to use the DSML on a regular base?
- What are typical use scenarios?
- What are the concepts of the DSML users are most interested in?
- What are aesthetic preferences of the DSML users?
- Is the group of prospective users homogeneous or rather heterogeneous with respect to the above questions?

Depending on the answers to these questions, further refinements may be required. If the prospective users do comprise not only experts but also novices that will use the DSML only rarely, it may be an option to provide a simplified, "light" version of the notation that covers only those concepts that are sufficient for inexperienced users. Note, however, that this may require substantial effort with respect to modifying/extending the syntax specification. If aesthetic preferences are expected to vary, one could provide different flavors of a notation, e.g., one that features artistic icons and a further one that emphasizes a more business-like style.

The design of an elaborate graphical notation recommends involving a professional graphic designer, which implies to somehow specify the corresponding assignment. Experience gained in previous projects indicates that example diagrams featuring a preliminary notation serve as a useful illustration for communicating concepts and modeling purposes to a graphic designer. Subsequently, the graphic designer is supposed to develop a draft version of the notation, which will then be—if required—repeatedly evaluated and revised until a satisfactory state is accomplished.

Finally, a documentation of the graphical notation is created. It consists at least of a comprehensive listing of all symbols and respective descriptions. Furthermore, it helps to illustrate the notation with a few example diagrams.

Input: (Revised) exemplary diagrams from requirements analysis.

Participants: Domain Expert, User, Language Designer, Tool Expert, Graphic Designer.

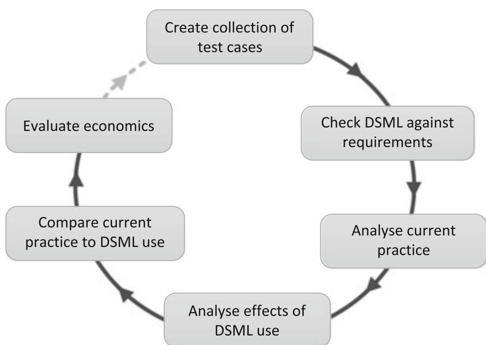


Fig. 9 Micro process "Evaluation and Revision"

Risks: Often, most of the participants will not have a specific expertise in designing and judging a graphical notation. While graphic designers should be familiar at least with the design of iconographic symbols, they may lack a clear understanding of the very purpose a DSML should serve. As a consequence, there is a clear risk that the graphical notation remains dissatisfactory. To reduce this risk, special attention should be applied to the selection of the graphic designer and to the execution of test procedures.
Results: Documentation of graphical notation, illustrated through exemplary diagrams.

5.6 Evaluation and Refinement

In order to ensure a certain quality level, it is mandatory to finally evaluate and possibly revise the DSML—in addition to the previous evaluation of the metamodel and the notation. The evaluation needs to account for specific challenges.

Objectives: creation of systematic and comprehensible evaluation; if needed, revision of tool.
Micro Process: First, the evaluation of a DSML and a corresponding modeling tool recommends checking them against the requirements—both generic (see Sect. 2) and specific. For some requirements it will be fairly easy to decide whether or not they are fulfilled. For example: "There should be concepts that allow for assigning probabilities to alternative paths of executing a business process". In these cases, it will be sufficient to determine and document whether or not a respective requirement is satisfied. With other requirements, an assessment may be harder. For example: "The modeling language should provide domain-specific concepts as long as they are regularly used and their semantics is invariant within the scope of the language's application". or: "The concepts of the language should be easy to comprehend by different groups of users". There are various approaches to reduce the respective uncertainty. First, it will usually be helpful to check language features against the background of a use

case. Therefore, evaluation recommends building on the use scenarios created for requirements analysis. Each use case serves to analyze whether and how corresponding requirements are satisfied by the DSML. For this purpose, a discursive evaluation is of pivotal importance. It should include language designers, domain experts, and prospective users. Users as the primary addressees of the language play a key role. However, that does not mean to simply rely on users' assessments. Instead, it is important to account for their respective background, such as experience with similar or other modeling languages, their formal education, the time they had to learn the language and/or the tool, their attitude towards DSML in general, etc. Similar considerations apply to other participants, too. However, language designers can be expected to have a different background which should enable them to provide more elaborate assessments—which, however, may also cause subtle bias in their judgment. Hence, the language we know is a necessary instrument, but it also limits our reasoning powers because we cannot entirely go beyond our language. Hence, it may well be that a different language that lies beyond our imagination would provide for a more effective and efficient structuring of the problem. At the same time the problem itself might appear different, if it is described/constructed in a different language. While these considerations seem to be of mainly philosophical nature, they are nevertheless important for the reflective evaluation of a DSML. On the one hand, they recommend being sensitive to the effect of existing languages skills. On the other hand, they emphasize the contingency of the subject: Language skills as well as language use are subject of change. Therefore a particular DSML is a moving target. As a consequence, its assessment should not be restricted to a certain point in time but rather be organized as a permanent process.
Input: Documentation of DSML, tool implementation, documentation, exemplary diagrams, use scenarios.
Participants: Domain Expert, User, Language Designer, Tool Expert.
Risks: The evaluation of languages may be jeopardized by various sources of sometimes subtle bias. In addition to that, there may be political interests involved that contribute to opportunistic assessments. Therefore, each assessment should be supplemented by a justification. If it is not possible to give a convincing justification, the assumptions an assessment is based on should be made explicit.
Results: evaluation report, revised and approved tool.

6 Conclusions

DSML seem as a logical evolution of GPML. Their emergence corresponds to the evolution of elaborate technical languages which are a key instrument of promoting industrial and post-industrial productivity. There would be no advanced craft, no engineering and no medicine without specific technical languages—just to name only a few fields. Against this background, it seems astonishing that complex systems are still analyzed and designed with GPML that are restricted to a few

generic concepts such as "class," "attribute," and "state". DSML represent a clearly more productive instrument for describing and analyzing problems as well as for designing systems. However, the specification of a DSML is a remarkable challenge. This is for various reasons. First, we are entering terra incognita to a certain degree, because so far there is only little experience with designing and using DSML. This results in the problem that it will often be difficult to ask the right questions. Second, the design has to account for competing or conflicting goals. One specific challenge is the quest for sustainability—even more than a model, a language should be stable for a longer time—which is contrasted by the contingent evolution of many domains. Third, the specification of a DSML requires a metamodeling language. However, it is not trivial to assess the benefit of an instrument one has little experience with for a complex task one is not too familiar with. The proposed guidelines are aimed at providing a framework and guidelines to reduce complexity and risk related to the development of DSML.

The guidelines have gradually evolved from our experience with designing DSML. In their current state, they represent a major improvement compared to the times when we did not have any methodical support. That does not mean, however, that we would regard them as mature. Instead, we rather see them as a repository of knowledge about developing DSML that should grow with the number of respective projects. In other words: Prospective users should not take all guidelines for granted, but reflect upon them and—if required—adapt them. This is especially the case for researchers who pursue the design of a DSML as a scientific task: Any research that either aims at analyzing a language and its use or at inventing new "language games" (i.e., artificial languages and actions built upon them) has to face a subtle challenge that is caused by the demand to justify prospective contributions to scientific knowledge: Every researcher is trapped in a network of language, patterns of thought and action he/she cannot completely transcend—leading to a paradox that can hardly be resolved. Designing a language is not possible without using a language. At the same time, any language we use for this purpose will bias our perception and judgment.

References

1. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software Syst. Model. **7**(3), 345–359 (2008)
2. Bunge, M.: Treatise on Basic Philosophy, vol. 3: Ontology I: The Furniture of the World. Reidel, Dordrecht (1977)
3. Fettke, P., Loos, P.: Ontological evaluation of reference models using the Bunge-Wand-Weber Model. In: Americas Conference on Information Systems (AMCIS), pp. 2944–2955. Tampa, FL (2003)
4. Frank, U.: Evaluating modelling languages: relevant issues, epistemological challenges and a preliminary research framework. Tech. rep., University of Koblenz (1998)
5. Frank, U.: The MEMO Meta Modelling Language (MML) and Language Architecture, 2nd edn. Tech. Rep. 43, University of Duisburg-Essen (2011)

6. Frank, U.: Some guidelines for the conception of domain-specific modelling languages. In: Nüttgens, M., Oliver, T., Weber, B. (eds.) Proceedings of the Conference 'Enterprise Modelling and Information Systems Architectures' (EMISA 2011), vol. P-190, pp. 93–106. GI, Bonn (2011)
7. Goldstein, R., Storey, V.: Some findings on the intuitiveness of entity-relationship constructs. In: Lochowsky, F.H. (ed.) Entity-Relationship Approach to Database Design and Querying, pp. 9–23. Elsevier Science, Amsterdam (1990)
8. Henderson-Sellers, B., Ralyte, J.: Situational method engineering: state-of-the-art review. J. Univers. Comput. Sci. **16**(3), 424–478 (2010)
9. Kelly, S., Tolvanen, J.P.: Domain-Specific Modelling. Enabling Full Code Generation. Wiley, Hoboken, NJ (2008)
10. Moody, D.L.: The "Physics" of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Software Eng. **35**(6), 756–779 (2009)
11. Odell, J.: Power types. J. Object Oriented Program. **7**(2), 8–12 (1994)
12. Opdahl, A., Henderson-Sellers, B.: Evaluating and improving OO modelling languages using the BWW-model. In: Proceedings of the Information Systems Foundation Workshop 1999, Sydney. www.comp.mq.edu.au/isf99/Opdahl.htm (1999). Last accessed April 2013
13. Süttenbach, R., Ebert, J.: A Booch metamodel. Tech. rep., University of Koblenz (1997)
14. Wand, Y., Weber, R.: On the deep structure of information systems. Inform. Syst. J. **5**(3), 203–23 (1995)
15. Weber, R.: Ontological Foundations of Information Systems. Coopers & Lybrand, Melbourne (1997)

Domain-Specific Languages and Standardization: Friends or Foes?

Øystein Haugen

Abstract Domain-specific languages (DSLs) capture the domain knowledge through the constructs of the language, but making a good language takes more than combining a set of domain concepts in some random fashion. Creating a good language requires knowledge not only from the domain but also from the domain of language design. Generic abstraction concepts turn out to be useful for many different domains and thus for DSLs. In this chapter we discuss how DSLs can benefit from standardized generic languages to cope with abstraction needs. A successful combination will keep the DSL simple and its implementation maintainable while the generic language will add expressiveness and structuring means. We give examples of DSLs as well as general ones and use the examples to illustrate our advice on how to make a good language. We share experiences of language evolution and finally show an example of combining a generic language for variability with a DSL for train signaling.

Keywords CVL • DSL • Language • Standardization • Variability

1 Introduction

We will introduce domain-specific languages (DSLs) and contrast them with the general programming and modeling languages (GPMLs). We look at how some of the GPMLs have evolved over years.

Ø. Haugen (✉)
SINTEF, PO Box 124 Blindern, NO-0314 Oslo, Norway
e-mail: oystein.haugen@sintef.no

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_7, © Springer-Verlag Berlin Heidelberg 2013

1.1 Domain-Specific Languages

A DSL is a language that is confined to a specific area (or domain). Very often the term is used to denote languages that are designed for well-defined, application-related domains. We should realize that what constitutes a domain is not easily defined. The domain may be company or project specific or it may be common to a larger group of experts or larger group of companies. A domain may be technically defined or dependent on market situations. The domains can be specific to one organization or company, or to a broader industrial realm. There is some evidence that success with DSL design is more often achieved if the domain is narrow and well bounded [1].

The advocates of DSLs argue that a DSL is the best way to capture the conceptual essence of a domain in a rather formal and manageable way. Some proponents will state that it is easy and quick to make a DSL and that there are tools on the market that generates tools directly from the language definition.

Skeptics of DSLs hold that creating languages is difficult, time consuming, and must be performed iteratively. While there are tools to help, they cannot conceal that when you make your own language you need to make all your supporting tools yourself, too.

Having created a DSL you have full control yourself. Typically you would like to make a code generator that can tailor exactly the low level code you need in your development. Proponents say that the code generator will be easy to make since the concepts are few and simple and you need not think in general terms neither on the source side nor on the target side.

A DSL can serve as a well-defined bridge between the domain experts and the IT-experts. The latter are software experts who know programming and modeling languages but lack the knowledge of the domain that is being described. Even other groups such as managers and customers may benefit from descriptions in a good DSL. A good DSL can sometimes be created from legacy notations that the domain experts have used for years. We show in this Chap. 1 such example by the train control language. Creating a DSL based on legacy notation then serves as a process of formalization as well as providing an iteration of the understanding of the domain even for the domain experts. This may even lead to discovering that old ways are obsolete and should be retired, or that old concepts need improvements, enhancements, or even simplification in light of new technology.

1.2 General Programming and Modeling Languages

To create synthetic languages is not new. In the dawn of computers any scientist with ambition would invent their own language with associated compiler. Just as there was a plethora of computer brands, there were numerous computer programming languages. They were domain specific and their domain was the

computer itself. Today we see these languages as general as they did not limit themselves to any application domain, but they had their specialties: COBOL [2] was for administrative computing while FORTRAN [3] targeted technical systems, LISP [4] should support artificial intelligence, and SIMULA [5] was made for simulation.

In the 1980s and 1990s both computer brands and computer languages consolidated and it became out of fashion for companies that earned their money from applications to make their own computer language as well. Standardization came in fashion and most programmers would use only a handful of programming languages. In the technical domain C and subsequently C++ provided what were needed and in the latest part of the consolidation period Java became the hottest language. In the same two decades computers found their way into all walks of life and programmers became a fairly large group of workers. The complexity of computer systems grew steadily and it was evident that computer systems could not be made directly by programming on the top of your head. This opened up for modeling languages and the consolidation of modeling languages resulted in UML [6]. UML did not satisfy everybody. The systems that were made to support the modeling and programming languages also grew like all other computer systems and many users found that their complexity outgrew their usefulness.

1.3 The Process of Making and Evolving a Language

After the turn of the century language design again became kosher and the modern DSLs started to appear supported by meta-tools such as MetaEdit+ [7]. Have these DSLs been successful? There are successes and there are failures with DSLs as there were among the early programming languages. We shall return to investigating what makes a good DSL, but it is clear that making a good language is not easy and it takes time and skill.

It takes time to make a general language, too. In fact languages need to evolve. When there is no evolution in a language it is dead. This happened to Latin and it has happened to programming language SIMULA and others. Some languages have survived and prospered. The modeling language SDL came in its first non-executable version in 1976 [8] and has followed a fairly steady pace with a new version every 4 years and it still exists, but its telecom domain is now more dominated by UML. UML itself came out of three ancestors and the conglomerate UML 1 was created. In 1999 it became clear that even modeling languages needed a precise definition and possibility to express executions. A major overhaul of the UML language began and in 2003 UML 2.0 [9] appeared as a result of the OMG[1] standardization process. Did UML 2.0 make all other languages obsolete? Some would state that UML 2 made new languages necessary [10] and argue that UML 2

[1] www.omg.org

is a language monster hardly possible to support by a tool. However, UML tools have appeared and they improve steadily proving that also standardized languages can be productive. The OMG has supplied a suite of models that tool vendors try to exchange and their efforts have been thoroughly walked through by the Model Interchange Working Group.[2]

This chapter sets out to investigate whether combining the standardized with the particular, the general with the special, could form an even better situation than choosing one of the approaches.

First we investigate why one would like to standardize, and then we explore when one would like to make a DSL. Finally we try to find how a harmony between the general and the specific can be achieved.

2 Why Standardize?

We all know that standards are useful. Travelers of the world struggle daily with a multitude of different sockets of electricity as well as differences in voltage as illustrated in Fig. 1. We also know that standards can be a pain in the neck and that there are more than one standard in a domain. A standard is often better for the user than the producer. This is not only a technical issue, but just as much a political and commercial issue. The business cases of adopting standards are different depending on your own actual position in the marketplace.

2.1 The Advantages of Standards and Standardization

A standard defines terms and an initial set of agreed interpretations. Standards therefore represent common grounds for people and machines. In this section we describe the advantages of standards and standardization on individual persons, teams, and on tooling.

2.1.1 Communication Between Human Beings

Referring to a standard easies the communication between persons who need to collaborate or understand each other. With the globalized economy complex products are made with pieces from all over the world and the pieces need to fit together. A standardized language help to define such pieces and how they are meant to fit together. In particular international companies need common grounds within

[2]http://www.omgwiki.org/model-interchange/doku.php

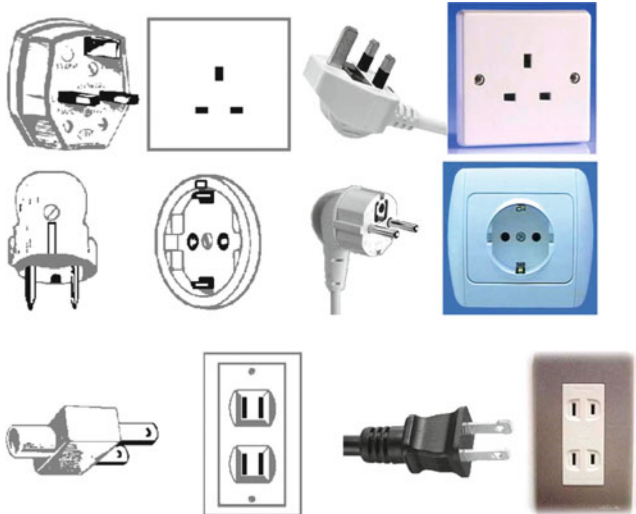


Fig. 1 Standards of electrical sockets (http://users.telenet.be/worldstandards/electricity.htm#plugs)

the company and standard languages for development is one way to minimize the internal confusion inherent to differences in culture and tradition.

Focusing on a common modeling language may also facilitate building competence needed to cope with the future. In the SISU project [11] we made a decision to focus on SDL [8] helping Norwegian companies in the embedded domain increase their automation and improve their methodology. Our strategy was multifaceted based on the following actions:

- Focus on one standard—SDL standardized by ITU in Z.100
- Take action to evolve the standard to meet needs discovered by the project—and this resulted in object orientation being included in SDL 92 and structured concepts in MSC 96 [12].
- Develop a methodology using the existing and upcoming standard—this resulted in the textbook Engineering Real Time Systems [13] supported by intensive 3-day courses for Norwegian industry.
- Prototyping tooling—in this case the OST tool [14] of object-oriented SDL showing that it can be implemented.

The SISU strategy was quite successful and in the end of the project a very large software system was ported from one of the industry partner to another one through the use of the SDL models. The commercial tools, however, took time to adapt to the enhanced SDL standard and this made it difficult to achieve the full benefit of the methodology.

The SISU methodology [13] is one example of another major benefit from standards, namely that teaching material and courses can be offered to a much

larger audience than if everyone was making their own language. This is clearly the case with UML now. Martin Fowler has explicitly described himself as a parasite on the UML standardization as he has made a series of popular books based on UML like [15], but he has never taken part in its creation or its standardization. This is of course welcome as the language designers may not be the best teachers or the best methodologists. The textbooks are in turn supporting courses. Almost all universities will present UML to their students, but the detail in which they teach UML vary considerably. Still UML does serve as a common ground for communication between system developers all around the globe.

2.1.2 Tooling and Portability

To transfer from one platform to another has been a challenge in systems engineering since the dawn of time. How different are the platforms? With the electrical example of Fig. 1 it is necessary to have a gadget that lets the current run from one plug to a compatible but different socket. Such a gadget may be only syntactical meaning that it only takes into account the pure construction of the plugs and sockets and makes sure that the electricity will flow through to the otherwise incompatible device. A syntactical converter for electricity works well these days because most portable devices can handle different voltage in their power supply. The voltage represents the semantics. You may still need a transformer from American 110 V to European 230 V if the power supply cannot do it.

The same overall picture holds for porting software. How different are the platforms? A standard may define a platform, but for years the definition of software platforms and languages has not been entirely precise, which is in fact quite difficult.

UML 2 had no formal definition and many scholars have tried to provide such a formal definition of UML 2 [16], but mostly partial semantics have been established and never has these semantics been used directly in the process of porting a UML model from one environment to another. Still it is easier to port a UML model from one place to another than porting a UML model into a Matlab environment. The latter obviously means making a transformer or compiler if you like.

Modeling languages may be more precisely defined than UML. SDL has had a formal definition for years [17] and porting SDL proved to be possible all the way to executable models. This corresponds to porting of programs written in programming languages. Porting of programs was not easy, either. FORTRAN programs behave differently depending on the layout of the memory and so do C programs. As a contrast Simula programs and Java [18] programs should behave consistently regardless of the memory layout as these languages are closed meaning that all reasoning can be done in terms of the language itself (and its runtime system).

Portability is one aspect of the more general tooling challenge. A potential advantage of standards is that it will motivate more companies and projects to create tools to support it. The motivation lies in the potential market as described in the subsection above on communication between people. But there are obvious challenges for the tool vendors that they persistently fail to meet. Tool makers

are notoriously conservative caring more about their current customer's short-term needs rather than putting efforts into implementing a new or upcoming standard. This has the risk that the tool makers miss the window of opportunity and the new standard may suffer and die. This could have been the fate of UML 2 as the tool vendors made their money from UML 1 tools (or SDL tools) and were reluctant to supporting the extensive UML 2 standard. Still there is an open question whether there are UML tools that support everything in UML 2. Lacking tool support effectively limits the systems designer and gives indications that the new standard is not good enough. This is one reason why the OMG has a process where there must be a tool implementation before the technology is considered "available."

It has also been an issue in the UML domain whether UML models could be exchanged between tools, edited and returned. This is the question of interoperability which goes slightly beyond portability. In reality it is very seldom that tools are interoperable enough to support a cloud of models where any tool can be used at any point in time. This is possible with simple textual descriptions, but hardly with UML models, but great improvements have been done even here and at the OMG standardization meetings sessions on tool interoperability take place regularly.

2.1.3 The Standardization Process

Sessions on interoperability are one example of how the standardization process and the standardization organizations contribute to common benefits.

The most important benefit of the standardization process itself is the quality assurance that it provides. While academic papers are reviewed but hardly discussed more than 5 min after the presentation, standards are debated for weeks, months, and years. This does not mean the end result is perfect, but it does mean that the drafts are significantly improved and that more views are applied to assessing the standard than what is common for academic papers.

While academic discussions are supposed to be exclusively about technical matters, standardization debates have strong influence on commercial interests. Often the tool vendors have a very clear opinion of what would be the easiest for them to implement, or they have already implemented a prototype tool and they do not want the standard to deviate too much from that. They want to be early in the marketplace and influencing the standard is one way to get a head start.

2.2 The Business Case of Standards

We mentioned above that standardization has a commercial dimension as well as a technical one. We shall take a closer look at what makes standards worthwhile for big as well as smaller companies.

Small companies will make small contributions, but these small contributions can be very important and serve as a seed that will grow and become something big.

To continue this botanical metaphor it is clear that the small seed needs the proper environment to prosper. Standards may form the proper environment for the small company contributions. The small company provides its contribution into a standardized environment and can thus benefit from whatever that environment can offer.

In the SISU project [11] we had this as our explicit strategy that focusing on SDL would give the small, Norwegian companies access to international tools, international competence, and international markets.

Big companies have several reasons for adhering to standards. Big companies do business with other big companies and following a supported standard makes such big business deals less vulnerable. Standards are normally supported by more than one tool vendor which means that if something goes terribly wrong there is an alternative. With standards more companies are dedicated to the persistence of the technology. Even when a standard does exist, the tool vendors will find means to keep their customers and in practice the users will find that they are closer attached to their tool vendor than they expected such that changing tool is a major endeavor.

Furthermore, big companies can apply an undergrowth of smaller companies that provide solutions to specific parts of the big company's portfolio of challenges, and these smaller companies can also be replaced because they adhere to the standard.

Supporting a standard makes it possible for tool vendors to invest more since they can count on the technology surviving for a longer time since more parties are involved in its survival. More investment in the tools is good news for the tool users regardless of whether they are big or small.

Finally, influencing or creating standards is an alternative way of dissemination of innovation results. Large research projects, e.g., under the European Commission have problems achieving impact that lasts beyond the end of the project. Fantastic prototype tools and stacks of papers may disappear from the public scene the day the project ends because there is no organization willing to support the results. Initiating a standard or evolving a standard is one way to contribute to the prolonged life of the project results.

3 Why Make a DSL?

The problem with adopting standards is that you need a way to distinguish yourself from your competitors. This is a dilemma that we see all the time in the marketplace; you need to have something that makes the customer buy your product and not your competitor that supports the same standard. One way to be distinguished from the common UML bunch is to create, support, and use a DSL. Why you would do that is what this section is all about.

Let us take an example from mobile phones. For many years Nokia was the market leader of mobile phones after having beaten (Sony) Ericsson by better functionality, no antenna, and long battery life. All mobile phones looked alike and basically worked alike. Even though both Ericsson and Nokia had used touch

screens, the keyboard was the preferred way of input. Then enter iPhone by Apple. The first iPhone was not the best phone as far as sound quality was concerned and it lacked much of the Nokia functionality, but it was a wonderful handheld terminal against the mobile Internet and the human interface distinguished it significantly from the Nokia/Ericsson tradition. iPhone deviated from the de facto standard of communicating with the device, but, on the other hand, iPhone decided to go along with the GSM standard rather than the American phone standard. iPhone became a winner and opened up a new niche in the market and the 3G network operators applauded the innovation. iPhone did not standardize its solutions in any open way. They patented their solutions and wanted to keep the monopoly of the new approach. But alas, enter Android from Google and Android is made open to the public and people are motivated to contribute and discuss it. Today (2012) Android terminals have overtaken iPhone and dominate the market while Nokia is struggling and trying to enter the Smartphone market[3] through collaboration with another strong monopolist, Microsoft. From this mobile phone historical scenario we can learn that standards do not last forever, that everything needs to evolve and that commercial success is partially connected with being special while still taking advantage of the established standards.

Described in language terms, the Nokia/Ericsson look-and-feel was a de-facto standard that the iPhone challenged. The iPhone look-and-feel with touch-screen, finger pinching, etc. was a DSL and Apple wanted it to stay a DSL. Google standardized it in Android and made it available to a larger group of vendors. Now they fight over whether the patents of the iPhone are infringed by Android.

3.1 Advantages of DSLs

The earliest domain-specific languages are defined just as legends of illustrations. Such notations were not only domain-specific, but in fact instance-specific as they were made up especially for the one illustration. Then more illustrations were made with the same legend and we can talk about a notation or a language.

One illustrative example of a synthetic language that has had considerable impact and that is still used with the same principles is the subway maps (Fig. 2).

A plain map is also a DSL as it represents a notation evolved over years where symbols are defined in the legend. Constructs like contour lines serve to convey the precise message of the terrain. The subway map took this one step further as it was recognized that there was no need for a close correspondence between the actual station position under ground and its position on the subway map. The important issue was the routes of travel that the network of lines represented. The diagrammatic approach was first invented by Harry Beck in 1931.

[3] http://finance.yahoo.com/news/worldwide-market-share-smartphones-220747882--finance.html

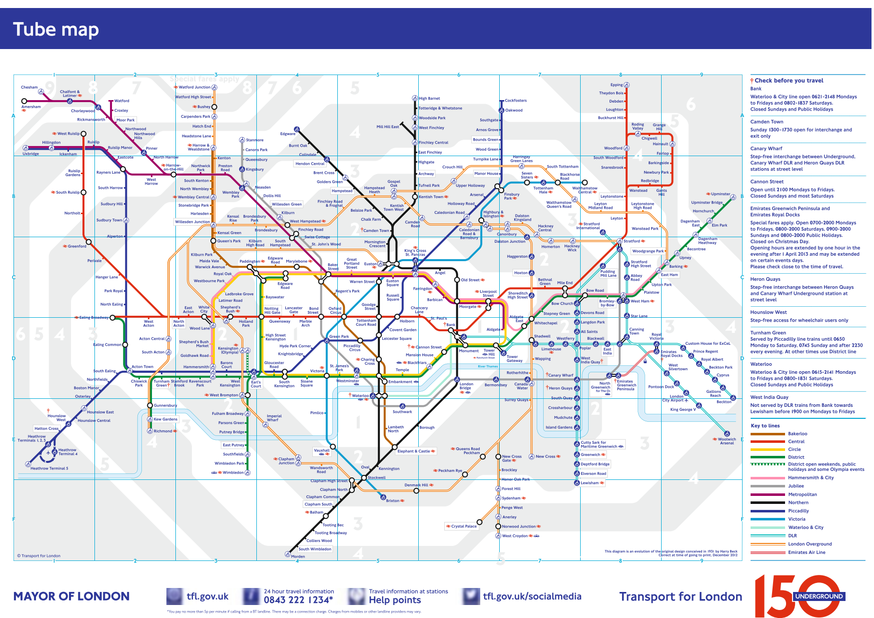


Fig. 2 London subway map (http://www.tfl.gov.uk/)

The subway map language has the core properties that we would like DSL to have:

- *Concise*: It captures the essence of a domain in a concise form
- *Formal*: It can be formalized such that automatic means can be used on it
- *Transparent*: The users find it intuitive and use it correctly

The language is concise when only the essence is defined and all that is in the description is relevant. That a language is formal we define to mean that it lends itself well to automation. The subway map can be formalized to support searches for shortest or fastest routes, etc. That the language is transparent is equally important. You do not need many years of schooling to read a subway map and act properly. In our systems domain you may need more schooling, but nevertheless we would like our languages to be readily available to its users. Very often a good DSL takes already existing notations from the domain as its starting point which immediately will let the domain experts recognize concepts.

A DSL defines the knowledge of a domain in a formal and concise way. It may be one of the best ways to capture the knowledge of a domain and making the DSL may be a good way to develop an understanding of the given domain. If the domain is confined to one company only, making a DSL will thus represent defining the technical knowledge of that company.

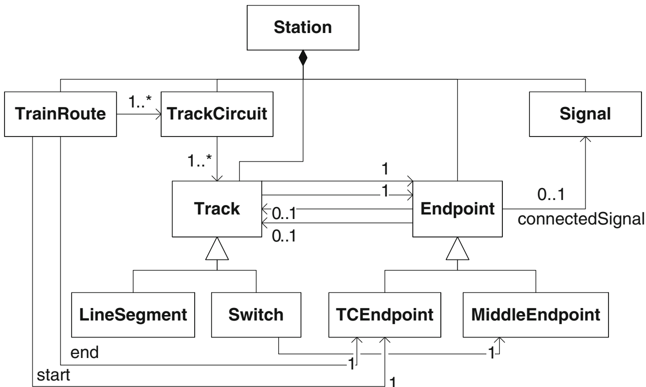


Fig. 3 TCL metamodel excerpts

Our three core properties are not the only way to categorize DSLs, but our intent here is to give a simple yet expressive way to characterize languages and not to capture all detailed aspects of a language or to compare languages. Please refer to [19] for a more comprehensive evaluation of DSL quality.

3.2 An Example of a Good DSL

Here we will present the train control language (TCL) [20–22] briefly and explain why we think this represents a good DSL. TCL is a domain-specific modeling language for modeling train stations and generating train station configuration code for train station signaling systems. TCL was created by ABB and SINTEF in the MoSiS project.[4] The purpose was to automate the development of signaling system source code. TCL is defined by a metamodel (Fig. 3), and an Eclipse plug-in has been developed constituting an editor and code generators.

As defined by the TCL metamodel, Station is the top concept, containing all the other concepts. A *TrainRoute* is a route a train must obtain before it can move into or out of a station. A TrainRoute is divided into *TrackCircuits*, which defines a certain amount of Tracks where a train can be located. A Track can either be a *LineSegment* or a *Switch*, which are connected by Endpoints. A TrainRoute starts and ends at a *Signal*, which will only give green light if the requested TrainRoute is available for the train. The concrete syntax of TCL is illustrated in Fig. 4 which also shows a view of the TCL editor made on the Eclipse platform using EMF [23] and GMF [24]. The code generators are described by MOFScript [25].

[4] http://mosis.reflector.os4os.org/modules/wikimod/

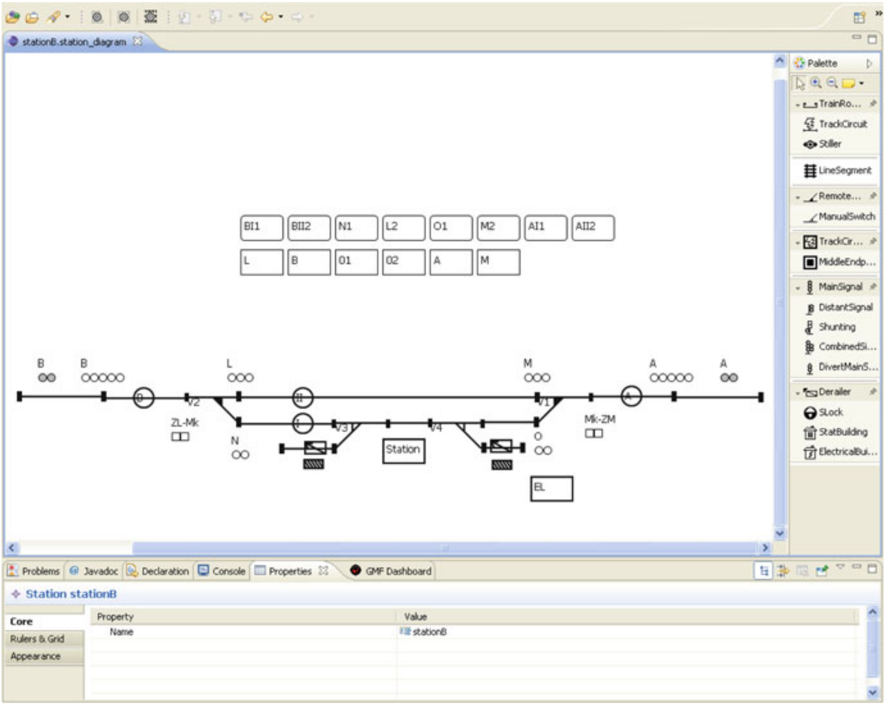


Fig. 4 TCL editor

The concrete syntax was based on the diagrams that ABB received from the Norwegian authorities when they were commissioned to developing the signaling for a new station.

3.2.1 Concise?

In assessing whether TCL is a good DSL we should investigate whether TCL is concise defining a small set of concepts that is still covering everything that is needed for its purpose, namely to define train signaling in a station.

The essential concepts are shown in Fig. 3 and it is evident that this is a rather small language. Specialization (inheritance) has been used to make the metamodel compact, and it also adds to a more concise understanding of the central concepts of train signaling. We should note that our illustration does not show the whole metamodel and that our language is not covering all concepts that would be needed to define all stations in Norway. Still the concepts are sufficient to define real stations and the code generation has been assessed against manually made code.

3.2.2 Formal?

The abstract syntax of TCL has been defined by the metamodel shown in Fig. 3 and this is clearly well defined.

The concrete syntax has been defined through GMF in our TCL tool. As it is embedded in a tool there is no doubt that it is defined, but there is no separate formalized definition of the concrete syntax.

The semantics of TCL are given by its code generators. There are three code generators giving three different kinds of formats that are used for different purposes in train signaling and the analysis of the train station. The code generators are made in MOFScript and acknowledging that MOFScript is a well-defined language, the TCL semantics is well defined.

3.2.3 Transparent?

The concrete syntax of TCL was based on existing diagrams produced by the Norwegian authorities who have been applied to define station signaling for several years. The TCL diagrams look extremely similar to the old diagrams but include also a few additions to define some of the structuring mechanisms.

Train signaling experts have read TCL diagrams and have had no problems understanding them. We have concluded that not much extra schooling is necessary for TCL to be used by its target users.

Even the produced code can be fairly easily assessed. The produced code is again very similar to the formats that have been developed manually before. This holds for matrices as well as the formulas. Indeed the code has been assessed by the experts and compared in detail with manually produced code.

3.3 The Business Case for Adopting DSLs

The main reason for deciding to go for making a DSL can be summarized by the following quote attributed most often to Einstein “Make everything as simple as possible, but not simpler.” This is exactly what a good DSL does. We mentioned above that technically a good DSL should be concise, formal, and transparent. From a pragmatic perspective we can say that a good DSL should be small and focused, easy to generate code from and capture the essentials of the company domain.

That DSLs are closely related to generating executable code from it, is not inherent to DSLs as such, but in practice this is a major reason for choosing to create a DSL. Many companies find that the commercial modeling tools are too complicated to configure or their code generation is not good enough for their purpose. There are several reasons for this.

First, general languages need general solutions and therefore the code that is produced may be more general than needed in any particular case. If the particular

case needs very compact or fast code, the general code generation may not be adequate.

Second, the particular system may have special interfaces to legacy code or proprietary hardware devices. The developers may want to control this legacy in detail and they want to describe this in transparent and concise ways.

Third, big tools are made by big vendors and big vendors are not always as attentive to smaller companies and their needs. Since big and general tools are more susceptible to errors and problems, it becomes attractive for a small company to choose a solution that seems to offer full control and relying very little on large tool vendors. (They must of course rely on the vendors of the meta-tool.)

4 Evolution of Languages

Natural languages evolve and artificial languages evolve for much the same reasons.

4.1 Evolution of Natural Languages

You need only read in a book from 50 years ago and you will realize that your native tongue has developed. Natural languages evolve for several reasons. The most obvious reason is that the actual world is changing. New concepts have been added or modified because new technology has been created, some natural phenomena have become more prevalent and new social patterns have emerged. There are multiple examples: 50 years ago we hardly talked about mobile phones or computers, we were not much concerned about the global warming or the greenhouse gases, and AIDS or terrorism had not entered our everyday vocabulary.

Some old concepts evolve and get new meaning. Take, for example, the term "boilerplate" which originated in the early 1900s to refer to the thick, tough steel sheets used to build steam boilers. Now a "boilerplate" is used to mean a template onto which we may add specifics having absolutely nothing to do with steel or steam boilers.

Furthermore, we have developed special natural languages for special purposes, almost like natural DSLs. One example of this is the language used in text messages or tweets. These languages have appeared since the message space was limited and the messages had special purposes and history.

4.2 Evolution of Synthetic (Artificial) Languages

Our synthetic and man-made languages for modeling and programming evolve for the same reasons as natural languages. New technology and new understanding

require new concepts in the description languages. Existing mechanisms can be improved inspired by usage experiences.

We may, however, say more about how synthetic languages evolve because they are made deliberately motivated by commercial or academic needs.

At first a DSL is simple and singular. The very inception of the DSL has been motivated by the opportunity to create something which will solve the problem at hand with a simpler language than offered by the general languages. The "infant" DSL only offers the most essential concepts and sometimes appears mostly as a proof of concept.

Then the DSL may be put into real use and we may call it the "youth" DSL. More concepts are added as the domain experts know that what they do not get into the language will not appear in the systems that are generated from the DSL. The language grows in number of concepts, but remains singular. A singular language means that it is intended to define one singular system, but possibly many times, each of which it defines the system from scratch. The TCL language (Fig. 3) is like this, there are enough concepts to define real systems, but each system needs to be described from scratch.

The "adult" DSL represents a major change in development. The DSL has had considerable success or else it would have died already. Unsuccessful DSLs are killed instantly as not much effort has been put into it. Thus we know our DSL is quite successful, it has users now who have not been part of the development and the number of DSL models has increased considerably. This is when the fact that the DSL is singular becomes a problem. Starting from scratch or from a pasted copy is not such of a problem when the number of models can be counted with your fingers, but if we are talking about 20 or 50 or 400, the need for mechanisms of reuse becomes interesting. The important observation now is that mechanisms of reuse very seldom is domain specific, and the decisions of what mechanisms to apply for reuse is not something a domain expert should take.

This is why the adult DSL is so critical. The rise in language complexity is often beyond the understanding of the language designers. They often continue the simplistic approach and grab hold of mechanisms similar to the macro, but fail to understand that macros have properties that do not make them well suited for defining reuse and that concepts like "type", "class," and "inheritance" may be much more sound.

The risk is that the language designers create something that they are not able to handle in their proprietary tools. The editors grow in complexity, but more serious is the rise in code generator complexity. Unfortunately, those mechanisms that have a long history of supporting reuse are still not trivial to implement. This is probably why so few DSLs have included well-established mechanisms such as inheritance and overriding. For general languages this is typically when such mechanisms are introduced after experiencing that the expressiveness of the language is not sufficient.

For DSLs there is another effect. Suddenly concepts that are not domain specific are introduced into the DSL. There are good reasons for doing so, but the improved DSL no longer look exactly like the legacy notation and it becomes an added

transparency challenge. Should we in TCL define inheritance of tracks? How should this then be shown?

If the synthetic language survives its adulthood, this is quite an achievement and most probably the language is now productive, but it is no longer very simple and it may no longer be as transparent as its infant version. Typically the adult DSL has more bugs in its code generation and more bugs in the supporting tools. This is the time when some of the users and domain experts will voice two related demands, namely a formal description of the language semantics and simplification. The demand for formal definition stems from experiences that it is increasingly difficult to get a common understanding of the models and the code generators are so complicated that they do not easily lend themselves to inspection. The demand for simplification has the same roots and would also make the desired formalization easier and sometimes since the language now has been used for a while, it may have been recognized that some mechanisms are not being used or are being used wrongly or dangerously.

Very few languages live through a simplification without suffering fatal blows. In fact this is typically when creating a new DSL (or general language for the same purpose) becomes the better option and the cycle starts over again.

To summarize, we recognize that the DSL evolution will normally face challenges that jeopardize the language properties: concise, formal, and transparent. In particular there is a critical stage in development when general concepts of structuring and reuse are demanded and often the language designers are not fully up to the challenge since implementing general languages is not a task for amateurs. We shall continue to investigate whether there is a possibility to combine the specifics of the young DSL with professional, standardized solutions of the general mechanisms.

4.3 The Evolution of Message Sequence Charts

Since company-specific DSLs are hard to document historically, I will share with you the history of MSC—Message Sequence Charts which is standardized by the International Telecommunication Union (ITU) in the Recommendation Z.120 [12, 26, 27]. I participated already before 1992, and from 1992 to 1996 I was the "associated rapporteur" for "structured concepts," and from 1996 to 2000 the main Rapporteur, which means the one responsible for the standard and the work related to it.

Sequence diagrams (or charts) had been in use informally in the development of communication systems in many companies such as Siemens, Ericsson, and Alcatel before 1990. The need for standardizing one such language came from a paper by Grabowski and Rudolph in 1989 [28] and the first standard for MSC was decided in 1992 [26]. In this "infant MSC" or Basic MSC as its core was later termed, we find

the domain-specific concepts Instance[5] and Message and a notion of a diagram. The most important semantic insight was that the messages were asynchronous which meant that sending and receiving must be understood separately. MSC-92 became quite popular and some reasonable editing tools were made.

The popularity of MSC-92 also meant that more institutions wanted to join in its future evolution and up to the "young MSC" (MSC-96 [12]) we had a fairly large working group consisting of a mix of academic scholars, tool vendors, and advanced industrial users. I was the one pushing structural concepts such as being able to define charts that could be referenced from other charts, and control structures ("inline expressions") such as alternatives and parallel composition. Furthermore, the academics from Eindhoven, Sjouke Mauw, and Michel Reniers insisted that we define a formal semantics (of MSC-92). We can say that MSC-96 represented a young adult in the terms of our classification in the former section. The language became more expressive through the inclusion of the general mechanisms, and the tool vendors failed to cope with it. The tool vendors made shortcuts and simplifying restrictions such that no tool really supported MSC-96 before MSC-2000.

Towards the MSC-2000 we achieved better understanding of the semantics of the structuring mechanisms and added concepts of MSC Documents (the context of several Message Sequence Charts) with object-orientation, as well as concepts for time and duration constraints and precise data. The latter can be seen as examples of domain-specific needs, while the former is an example of necessary general structuring mechanism. MSC-2000 [29] was a grown-up language and the tool vendors still did not quite live up to it. Possibly MSC-2000 was too advanced and coincided with the telecom users were drifting away from SDL and MSC to UML where another dialect of Sequence Diagrams had been included in its UML 1.x version. May be MSC-2000 was an old language and time was ripe for a new iteration?

From 1999 I moved from ITU to OMG and started working to merge MSC into UML. While MSC was a language in its own, Sequence Diagrams is a sub-language of UML and the challenge was to integrate it with the other sub-languages. UML 2.0 saw the integration and the most central concepts of MSC is now in UML [30]. Even though Sequence Diagrams are the second most used diagram type in UML, the more advanced mechanisms are more seldom used and opinioned designers have voiced that they are too complicated. Finally, after almost a decade after UML 2.0 appeared tool vendors support most of Sequence Diagrams.

5 Harmonizing the General and the Specific

There are two related distinctions that we investigate in this chapter. One is between the domain specific and the general purpose and the other is between the proprietary and the standardized. These distinctions are related since it is not reasonable to

[5]MSC Instance corresponds to UML Lifeline

standardize a proprietary language and general languages are often in some way standardized either formally through a standardization organization or informally ad hoc or through market dominance.

5.1 Standardizing DSLs?

One supposedly easy way out of the conflict between standard and domain specific could be to standardize the DSL. This is being done all the time. Many would argue that this is mostly what OMG is doing. There are language standards and reference models for almost all corners of engineering and technical competence. Of course when a language is standardized much of its versatility and dynamical evolution disappears while, on the other hand, it becomes more widespread and will probably be considered more serious and long-lasting.

5.2 Combining DSLs and General Purpose Languages

Realizing that our challenge lies in harmonizing the general with the specific such that their respective advantages are maintained through evolution we discuss three different ways to combine the general with the specific.

5.2.1 Ad Hoc Combination

Like natural languages evolve in an ad hoc way in the streets, one may incrementally evolve a DSL with general as well as domain-specific concepts in some random fashion. Chances are, however, that the resulting language will reflect this process and be perceived as increasingly confusing and difficult to implement.

Still it worked for English as the world language. English is a potluck dish of Norse, Anglo-Saxon, and Latin terms spelled in ways that have only slight resemblance to how they are pronounced. Nevertheless, English serves as the standardized natural language. The world never chose Esperanto [31] even though it is cleaner and simpler. Many would say the same for UML as we just did for English.

We mentioned earlier that DSLs do evolve from the definitely domain specific to the more general concepts following their immediate success. Their success may continue, but the competence of the language designers and implementers is increasingly challenged. The challenges are not linear, for every new concept added its relationship to every existing construct must be (in principle) considered and unintentional inter-play between constructs are more and more likely to occur. This is why DSLs that remain successful are often fairly simple. DSLs that go beyond

the purely domain specific can be on their path to becoming a standardized language (for a certain rather wide domain). We have seen DSLs originating from research projects that aspire to becoming standards like EAST-ADL.[6]

As the complexity grows and the ratio of general terms grows, an alternative approach to combining the domain specific with the general becomes more and more attractive. This is what we shall describe next, namely to amalgamate the domain-specific concepts into an existing general language. For EAST-ADL this is shown by the fact that it exists also as a UML profile.

5.2.2 Amalgamate a Standard Language with a DSL

Amalgamation can be done systematically or ad hoc. UML profiles represent this approach where the starting point is a standard language UML and more concepts are added through "stereotypes" which are defined in profiles. The stereotypes are in fact new concepts that in principle suck up properties and capabilities from general mechanisms and extend these capabilities further.

It is not entirely clear from the definition of the profile mechanism how much the stereotyped concepts need to suck up from the general concepts. Profiles often appear as language extensions of UML but where the connection to UML is only partially maintained. Furthermore, profiles often have a plethora of concepts that are not properly combined with the UML base. The intention of profiles is clear. It was intended to give the users a way to extend and enhance the (UML) language such that the UML tools are still able to read and handle the profiled descriptions. The syntactic extensions defined by profiles are easily handled by most UML tools, but the semantics is a different story. Normally, profiles are used in special contexts and there is no real need for complete generality in the language definition. The stereotypes serve as programming language pragmas and the code generator acknowledges them eagerly. It has been said that almost no real use of UML exists without added profiles. At least this is probably true for UML used as a programming language. A big problem of course is that of interoperability. As long as the code generator is proprietary the exchange of profiled UML descriptions is merely an exchange of syntactic elements.

Amalgamation can also be done by merging systematically metamodel packages of the general notions and the domain-specific ones [32]. As with profiles such amalgamated metamodels can be hard to overview and either their semantics is clear, but not intuitive, or the syntax is clear and the semantics rather incomplete. In short such approaches may suffer from transparency problems even when their approach is systematic and formal.

[6] http://en.wikipedia.org/wiki/EAST-ADL

5.2.3 Separate Descriptions

Finally, we can define an approach where the general and the specific components are maintained entirely separate. The combination is well defined through the definition of a small set of interfacing concepts. Typically the general description will define transformations on the specific description representing a variant of generative programming [33]. The approach may be summarized in these bullet points:

- A complete description contains a pair of descriptions where one description is in a generative language and the other is a domain-specific one.
- The generative description refers to elements in the domain-specific description
- When executing the generative description the result is a modified domain-specific description.

The generative description is given in a generic and standardized language with a clear semantics and we expect tools to exist that will support its use. The domain-specific description, on the other hand, is dedicated to covering terms and concepts defined by the domain experts. The DSL should be able to describe every singular system in the designated domain. The execution of the generative language will result in one or more new DSL descriptions.

5.3 Standardizing a Generative Language

In this section we present the challenges and opportunities of standardizing a generic language which is intended to be used together with other languages and thus not as a stand-alone language. Furthermore it is intended to be used with a number of other languages and even languages that are not yet known.

We shall highlight the challenges and opportunities of using a general generative language together with a domain-specific one through an example. The example is the common variability language (CVL) as it was conceived in the MoSiS project [34] and input to OMG standardization [35]. Here our focus is on the challenges and benefits of standardizing a generative language and not to argue that CVL is the only possible approach to defining variability.

5.3.1 Introducing CVL: The Common Variability Language

CVL is a generative language intended to describe variability. Thus, combining a CVL description with a description in a base language would actually describe a possibly infinite set of models in that base language. The combined description is a description of a product line. In Fig. 5 we show how the OMG Request for Proposals [35] show the combination of a CVL description and a base model.

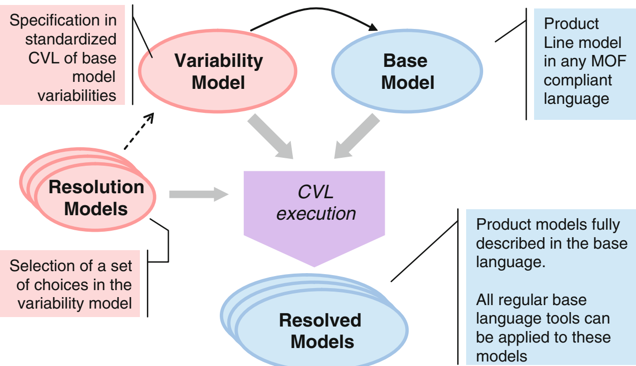


Fig. 5 The combined descriptions of CVL

The Variability model contains descriptions of how the base model can vary in precise terms. It ends up with specifying how the base model is transformed into a new model in the same language as the base model. The leaf nodes of the variability model are substitutions that exchange some parts of the base model with other parts. The Variability model also contains a version of feature models originating from Kang [36] and enhanced by, e.g., Czarnecki [33] as shown in Fig. 6. The resolution models define how the features are resolved, what choices that are made. The resolution, variability, and base models are brought into a CVL execution and resolved models come out as a result. In product line terms these resolved models represent individual products of the line.

5.3.2 Challenges to Such Generative Language

The first challenge is that of defining a standard language that can relate to any other language (that defines the base model). The solution to that challenge is to step one meta level up and demand that the base language is defined in MOF such that CVL can refer to the base model as MOF objects and their internal relationships as MOF references. By applying MOF reflection we can still solicit properties of the base model useful in our tooling. In our MoSiS CVL Tool we apply such MOF reflection to match placements with replacements in the fragment substitution operation. The tool checks the DSL-specific types of the MOF references through reflection and make sure that these types match pairwise between the placement and the replacement.

The second challenge is about concrete syntax. The solution to the first challenge only provides a connection inside the computer. How can the users of CVL and the base language see what is present in the models? Clearly the base language has already its own concrete syntax and preferably the CVL concrete syntax should

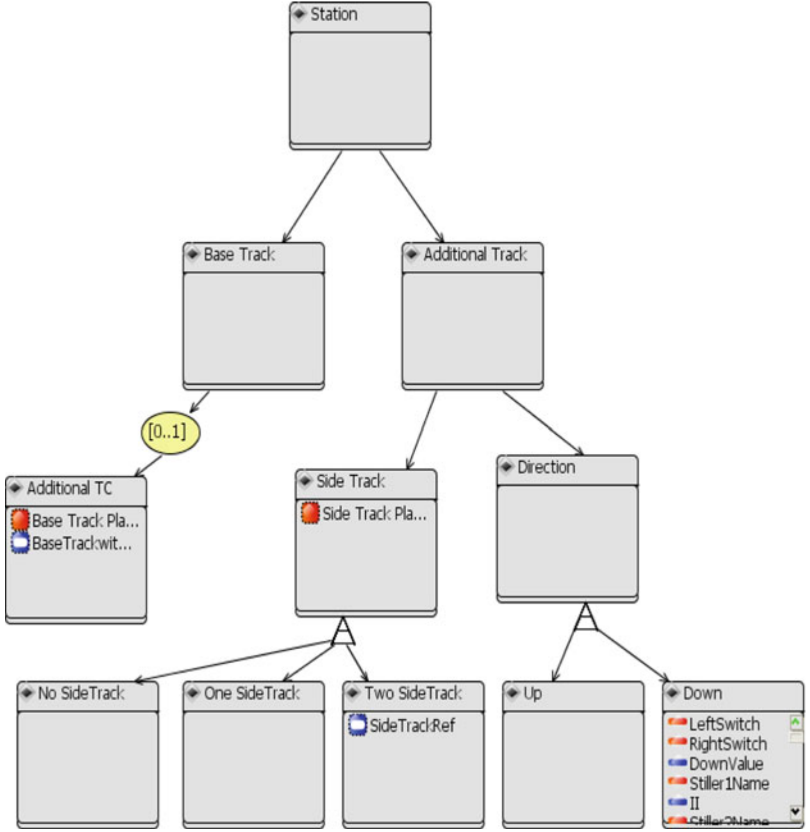


Fig. 6 MoSiS CVL concrete syntax example of the abstraction layer model

match that. This is not as simple as with the metamodels, there is no abstract level of concrete syntax on which the CVL concrete syntax can be defined.

The combination of CVL and the other DSL is related only to those elements that include references from CVL to the DSL objects. CVL also includes an abstraction layer similar to the traditional feature diagrams and for that we may define a CVL specific concrete syntax as shown in Fig. 6.

For the realization layer of CVL that represent the association between CVL and the base DSL, it is quite clear that some kind of visual amalgamation must be applied. One approach would be to introduce stereotypes in the concrete syntax similar to what had been the case when using profiles. An extension to this approach would be to provide special symbols to emphasize the stereotyping, or special visual effects like colors or line styles or thickness of lines. This is feasible, but possibly not always practical or visually pleasing. Traditionally, we think of such stereotypes as being static since they only represent elements of a combined description which

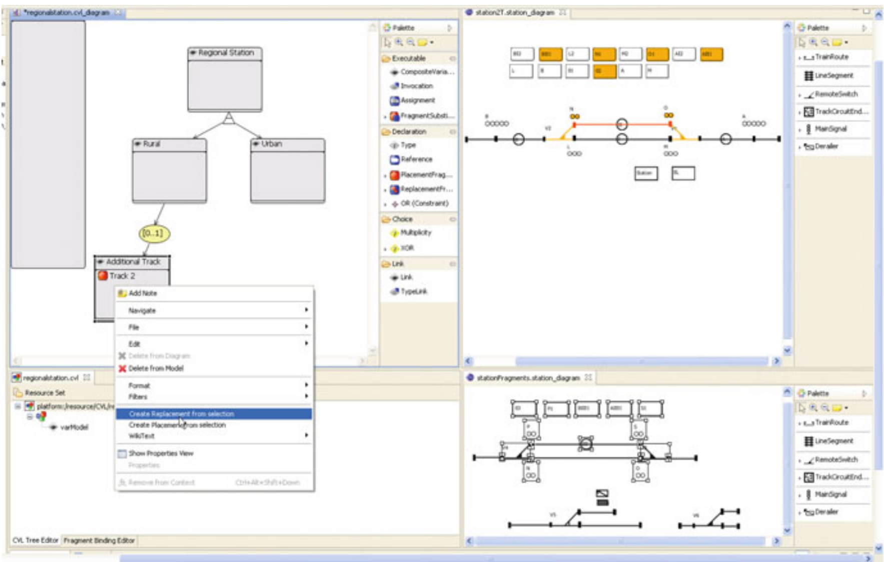


Fig. 7 CVL Tool interfacing to a TCL base model

is indeed static. This may quickly lead to descriptions that are rather overloaded with effects and/or text.

We therefore suggest considering the combination of CVL with another language as something dynamic. All syntactic effects overloaded on the base model at once may become too much. Since we normally never print out complete descriptions any more there is really no need for this complete concrete syntax. Our solution is therefore to leave the CVL concrete syntax for the realization layer to the tooling, but to provide a simple interface that tools should realize. MoSiS CVL has defined four interface functions, two for selection of objects and two for highlighting of objects.

In Fig. 7 we show how this works. There are three graphic panes on this screen. The left part represents the CVL description and the right side represents the DSL description. In our case the DSL is a TCL [20, 22] which represent the signaling of a train station. In the lower right pane we show that some objects are selected. In the CVL feature diagram on the left we have right-clicked on an object to get a context sensitive menu. The screen shot shows when the user is creating a replacement fragment from the selection. A "replacement fragment" is one of the elements of CVL that represent the realization layer referring from CVL to the DSL description. The CVL object will refer to the set of TCL objects. The upper right pane shows the visualization of such a connection between a CVL object and the TCL description. Objects of TCL are highlighted in red and orange.

When we combined the CVL Tool with the TCL tool we chose to realize the highlighting functions by applying colors. We could have chosen differently if

colors had already been applied within TCL. Then we could have applied blinking or even dynamically added stereotypes. In this way we have converted pieces of the concrete syntax into features of the tool.

From a standardization point of view we may ask how such combinations should be standardized. The CVL Tool has four interface functions. Should they represent the concrete syntax standard of the realization layer of CVL? Does this approach mean one stop towards standardizing tools supporting a language rather than merely the language? By standardizing these functions any language tool that realizes this interface will work seamlessly with any CVL tool (that applies these functions). Thus, this does serve one purpose of standardization since we achieve a higher degree of interoperability between tools supporting the same standard.

The third challenge is to keep the generative language general, but still sufficiently powerful and pragmatic. It is tempting to construct special functions of the generative language that takes into account specific knowledge of the associated language. In our example with the combination of CVL and TCL, it may be tempting to introduce variability mechanisms that only apply to train station signaling. Such mechanisms could be to take advantage of TCL dependencies such as that whenever a track circuit is removed all signals associated with it will also be removed. The benefit could be a more domain expert friendly variability language and a more effective way to define product lines of train stations. The downside is that CVL is no longer fully general. Either we get variants of CVL or we need to generalize the domain-specific mechanism.

We may compromise by applying the same technique as for concrete syntax, namely to specify an interface applied by CVL and realized by the DSL-specific tools. Again, this interface could be standardized, but it is much more difficult than standardizing functions of selection and highlighting. The latter concrete syntax functions have little effect on the models as such. They represent visualizations. Any function defining the cascading effects of a CVL operation on a DSL model, however, will be interfering severely with the modeling. There may be disagreement about the dependencies. In our example, maybe not all train signaling engineers agree that the signals should be removed when the track is removed. Since the definition of the interface must be fully general since it would be part of the definition of the generative language, there would be a need for another standardization process for defining the DSL-specific realizations of that interface if full interoperability shall be achieved.

5.4 *Opportunities of the Standardized Generative Language*

We have seen that there are definitely some challenges to defining a standard generative language such as what we are doing with the CVL. There must be some opportunities and upsides to going down this path of two languages rather than one amalgamated language.

The first opportunity is related to separation of concern. The CVL concerns itself with defining the variability and therefore the DSL may concern itself exclusively with the domain-specific concepts. When the needs for reuse and variability occur, this is left to the accompanying generative language CVL. The obvious effect of this is that defining the DSL is simpler. The designers of the DSL can concentrate on the domain specific which is their expertise. Defining the DSL often also implies implementing the code generator representing the operational semantics of the DSL. A simpler language is also simpler to code generate from. Again, the DSL designers and implementers are trained in the code coming from the domain-specific terms, but less trained in implementing general constructs of reuse and variability.

The second opportunity is related to the first one and related to standardization. A standard language will also have supporting tools. Most often there will be several tools supporting the standard and those using the DSL may choose freely between those tools supporting the accompanying generative language (such as CVL). There may even be an open-source or community tool that is free of charge on which you can experiment while assessing whether the DSL approach is viable for your project or company.

The third opportunity can be seen as a continuation of the two former opportunities. Separating out the variability (or any other functionality that can be solved by a generative language) may help the DSL development to become more agile. Since no successful DSL stays constant there will be demands for improved flexibility from the DSL users. A standardized generative language may serve as an immediate solution to the user demands and a reference for more elaborated solutions later. It may be that the train signaling language should include module or type concepts later which would take care of some of the functionality that the combination with CVL takes care of today. Since general, generative languages (such as CVL) work well with all languages, they would continue to work well with the augmented language, too. Thus the DSL development using generative accompanying languages and tools are fairly future (forwards) compatible. The intermediate solution with the accompanying generative language gives high functionality with little effort since the standard language is supported by an existing tool. Later assessment may even conclude that the combined solution serves well enough.

6 Discussion and Conclusions

The general and the specific are not foes and not the beauty and the beast. In fact DSLs may go well with generative languages that define aspects of reality in a crosscutting way.

In this chapter we started by presenting a view on DSLs and synthetic languages in general. We emphasized that all languages must evolve and that creating languages is not trivial even when you know the domain well.

We then presented advantages and challenges with standardization and hopefully were able to convey that standardization have advantageous properties that may not be obvious to everybody.

Recognizing the potential benefits of being particular and making a DSL as well as going for standards, we investigated whether combinations and compromises could be found.

We have presented the combination of the CVL with a DSL of train signaling (TCL). Our experiences with this combination and of some other combinations that we have attempted during the MoSiS project are that this works well. The DSL designers are happy to focus on the purely domain specific and the end-users are able to apply the CVL Tool after very moderate tutoring. We have also experienced that the combination can serve as exploration of more language specific solutions to variability. This holds for variability terms associated with SysML [37] and UML.

There are two questions that arise naturally: (1) is variability the only crosscutting generative aspect that lends itself suitably for standardization? and (2) why not go all the way and just apply a general transformation language such as QVT [38]?

Let us discuss the last question first. Is there any need for a new standard generative language for (say) variability when there is already the standard QVT? This is very similar to the discussion about general description languages versus DSLs. QVT is general relative to the transformation domain which means it is comprehensive and not directly available to those without background in the transformation field. CVL is domain specific to modeling variability and there is a set of dedicated mechanisms that should be comprehensible for those familiar with product line concepts. This is the main reason, but we can also add that CVL is general enough to describe a transformation into whatever target product you want. In other words CVL can define the difference between any two descriptions in the accompanying DSL [39] and as such is comparable with QVT expressiveness.

If we accept the desirability of designing DSLs, we could formulate the first question above into whether it is possible to design a generative DSL for any general crosscutting field. We do not have a list of such general crosscutting fields, but it is quite clear that it would be possible to make a DSL for expressing debugging instrumentation, or logging for that matter, to take two examples well known from aspect-oriented approaches. As with transformations in general we will soon enter into problems concerning the order of applying the generative descriptions. Obviously there is a limit to how well this approach works if there are many combined descriptions. Our experiences are limited to combining the variability language CVL with different base languages.

We have shown that there are challenges to the combination of base languages with CVL, but that there are also many challenges to making a complete DSL that incorporates the variability constructs. Our combined approach is an agile solution that keeps the general from the specific and makes it possible to standardize the more complex general constructs.

References

1. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 471–480. ACM, Waikiki, Honolulu (2011)
2. Conference on Data Systems, L. COBOL – 61: report to Conference on Data Systems Languages, including extended specifications for a common business oriented language (COBOL) for programming electronic digital computers. In: Conference on Data Systems Languages Maintenance Committee. Department of Defense, Washington (1962)
3. ANSI: USA Standard FORTRAN: Approved March 7, 1966, p. 36. American Standards Association, New York (1966)
4. Weissman, C.: LISP 1.5 Primer. Dickenson Publishing Company, Inc., Belmont (1967)
5. Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., Nygaard, K.: SIMULA BEGIN. Petrocelli/Charter, New York (1975)
6. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, pp. XVII, 550s. Addison-Wesley, Reading (1998)
7. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation, pp. XVI, 427. Wiley, Hoboken (2008)
8. ITU: Z.100. In: Faergemand, O. (ed.) ITU Specification and Description Language (SDL), p. 237. ITU-T, Geneva (1993)
9. OMG: Unified Modeling Language 2.0. OMG, Needham (2004)
10. Greenfield, J., Short, K.: Software Factories, p. 666. Wiley, Indianapolis (2004)
11. Haugen, Ø., Bræk, R., Melby, G.: The SISU project. In: Proceedings of the Sixth SDL Forum, SDL '93 Using Objects. North Holland, Darmstadt (1993)
12. ITU: Z.120. In: Rudolph, E. (ed.) Message Sequence Charts (MSC), p. 78. ITU-T, Geneva (1996)
13. Bræk, R., Haugen, Ø.: Engineering Real Time Systems. In: Welland, R. (ed.) BCS Practitioner Series, p. 398. Prentice Hall International, Hemel Hempstead (1993)
14. Hauge, T., Haugen, Ø.: OST—an object-oriented SDL Tool. In: Forth SDL Forum. Lisbon, Portugal (1989)
15. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, pp. XXX, 175. Addison-Wesley, Boston (2004)
16. Broy, M., Cengarle, M.: UML formal semantics: lessons learned. Softw. Syst. Model. **10**(4), 441–446 (2011)
17. ITU: Z.100 Annex F. In: Olsen, A. (ed.) Specification and Description Language (SDL) Annex F. SDL Formal Definition, pp. (33+437+183). ITU, Geneva (1993)
18. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley, Reading (1996)
19. Mohagheghi, P., Haugen, Ø.: Evaluating domain-specific modelling solutions. In: Trujillo, J., et al. (eds.) Advances in Conceptual Modeling—Applications and Challenges, pp. 212–221. Springer, Berlin/Heidelberg (2010)
20. Endresen, J., Carlson, E., Moen, T., Alme, K-J., Haugen, Ø., Olsen, G.K., Svendsen, A.: Train control language—teaching computers interlocking. In: Allan, J., Arias, E., Brebbia, C.A., Goodman, C., Rumsey, A.F., Sciutto, G., Tomii, N. (eds.) Computers in Railways XI (COMPRAIL). WIT, Toledo (2008)
21. Svendsen, A., Møller-Pedersen, B., Haugen, Ø., Endresen, J., Carlson, E.: Formalizing train control language: automating analysis of train stations. In: Ning, B., Brebbia, C.A., Tomii, N. (eds.) Comprail 2010. WIT, Beijing (2010)
22. Svendsen, A., Olsen, GK., Endresen, J., Moen, T., Carlson, E., Alme, K-J., Haugen, Ø.: The future of train signaling. In: MODELS2008. Springer, Toulouse (2008)
23. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/. Accessed 25 Apr 2012

24. Eclipse Graphical Modeling Framework (GMF). http://wiki.eclipse.org/GMF_Documentation (2009). Accessed 25 Apr 2013
25. Oldevik, J.: MOFScript eclipse plug-in: metamodel-based code generation. In: Eclipse Technology Workshop (EtX) at ECOOP. Nantes (2006)
26. ITU: Z.120. In: Rudolph, E. (ed.) Message Sequence Charts (MSC), p. 36. ITU-T, Geneva (1993)
27. ITU: Z.120. In: Haugen, O. (ed.) Message Sequence Charts (MSC), p. 126. ITU-T, Geneva (1999)
28. Grabowski, J., Rudolph, E.: Putting extended sequence charts to practice. In: SDL '89—The Language at Work. SDL Forum 1989. North-Holland, Lisbon (1989)
29. Haugen, Ø.: MSC-2000 interaction diagrams for the new millennium. Comput. Netw. **35**, 721–732 (2001)
30. Haugen, O.: Comparing UML 2.0 Interactions and MSC-2000. In: SAM 2004: SDL and MSC Fourth International Workshop. Springer, Ottawa (2004)
31. Zamenhof, L.L.: Tipo-Litographiya H. Keltera. Unua Libro, Warsaw (1887)
32. Perrouin, G., Vanwormhoudt, G., Morin, B., Lahire, P., Barais, O., Jézéquel, J.-M.: Weaving variability into domain metamodels. Softw. Syst. Model. **11**(3), 361–383 (2012)
33. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications, p. 864. Addison-Wesley Professional, Reading (2000)
34. Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Svendsen, A., Zhang, X.: Standardizing variability—challenges and solutions. In: Ober, I., Ober, I. (eds.) SDL 2011: Integrating System and Software Modeling, pp. 233–246. Springer, Berlin/Heidelberg (2012)
35. OMG: Request for Proposal. Common Variability Language. Object Management Group, Needham (2009)
36. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1990)
37. OMG: SysML-OMG Systems Modeling Language. OMG, Needham (2010)
38. OMG: Query/View/Transformation, v1.1. OMG, Needham (2011)
39. Zhang, X., Haugen, O., Moller-Pedersen, B.: Model Comparison to Synthesize a Model-Driven Software Product Line. In: 15th International Conference on Software Product Line Conference (SPLC), Munich, 2011

Domain Engineering for Software Tools

Tony Clark and Balbir S. Barn

Abstract General purpose software engineering tools are expensive to develop and maintain, and often difficult to learn and use. Domain-specific tools tend to be small, focussed and easier to learn; however, domain-specific tooling tends to be technology specific and therefore introduces interoperability problems. This chapter provides a contribution to DSL tool development by describing a *language-driven* approach to domain engineering whereby a tool is modelled in terms of the syntax and semantics of the language it supports. This chapter uses UML to define a simple class modelling language and its tooling.

Keywords Domain specific language • Meta-modelling • Software tools

1 Introduction

The proliferation of methods in the 1980s and early 1990s yielded to the overwhelming force brought about by the development of a unified language: UML and its variants. The adoption of a single language provided the economic incentive to produce supporting tools. A number of commercial tools emerged, but perhaps because of the general purpose nature of the language, it rapidly became apparent that development, support and marketing of such tools was no small task. Many tools dropped away leaving a small number of commercial players trying to support a very diverse market. The result is that tools to support UML-style development are like Swiss-Army Knives: they aim to support all possible tasks with little guidance.

Software tools and technologies fall into two broad categories: *horizontal* and *vertical*. A horizontal technology can be applied across a wide range of application

T. Clark (✉) · B.S. Barn
Middlesex University, London, UK
e-mail: t.n.clark@mdx.ac.uk; b.barn@mdx.ac.uk

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_8, © Springer-Verlag Berlin Heidelberg 2013

areas because it provides general purpose functionality. A vertical technology is specific to a particular application domain. Perhaps as a reaction to the lowest-common-denominator (LCD) flavour of UML, there has been a rise in interest in Domain Specific Languages (DSL) and their associated tools. Unlike UML which is a horizontal technology, a DSL aims to be a vertical technology by providing a means to represent, analyse and manipulate concepts from a specific domain such as real-time systems, telecomms and transport networks.

DSL tools can afford to be simpler than the UML equivalents because they expose very specific functionality. However this simplicity comes at a price. Where UML aims to support all aspects of the software development process and therefore can guarantee interoperability, the DSL approach leads to tool chains that can have interoperability problems [6]. In addition, because UML adopts LCD, it is possible to use it in a wide range of diverse domains. Typically each software engineering project is unique and therefore a DSL approach leads to a new toolset for each project.

The problem to be addressed is how to gain the benefits of both approaches. UML represents a universally applicable, interoperable, cost-effective technology that suffers from complexity and insufficient support for project domains. While attempts to use the stereotyping capability within the UML standard are numerous, tool capability is limited. DSLs produce technologies that are simple and focussed but suffer from complexity of development and interoperability issues. We would like to be able to produce DSL tools *in a standard way* that is universally applicable, leads to tool interoperability and therefore makes the development of DSL tools cost effective without leading to huge one-size-fits-all platforms.

Our proposal is to apply domain engineering techniques at the tool level using a standard tool meta-language. A tool model expressed in the tool meta-language can take two forms: a *specification* of the required tool and a tool platform *configuration* model. The former contains a model of the domain and specifies tool functionality over the domain, the latter is consistent with the specification and can be uploaded into any (proprietary or open-source) suitable framework that will configure itself appropriately.

Domain Engineering of Tool Models, as envisioned by our proposal, would be a phase of any new software engineering project. Tool models from similar projects could be used as a starting point, and modelling techniques such as transformations, slicing and merging can be brought to bear. Tool specifications would be useful as a teaching aid, and model slicing could be used to produce minimal sub-sets whose functionality could be gradually increased to produce an incremental training programme.

Tool models must contain a variety of features in order to be effective. Each domain can be considered as a language consisting of syntax and semantics. The tool itself can also be considered as a language whose syntax involves menus, editors, tree-browsers, etc., and whose semantics involves events that cause changes to the application domain. A model will describe how the state of a tool is serialized in order to guarantee interoperability. Models should also contain non-functional aspects such as usability and efficiency.

Our claim is that it is possible to take a domain engineering approach to tool modelling. A tool modelling standard would require a collaborative effort involving experts from a wide variety of disciplines. This chapter validates the claim by providing an overview of a simple meta-language for specifying tool models, using the language to specify a simple tool, and finally analysing the model in terms of the resulting benefits.

This chapter is organized as follows: Sect. 2 describes related approaches to developing modelling tools; Sect. 3 describes our approach to applying language engineering to tool development; Sect. 4 defines a class modelling language using the proposed language-driven approach; Sect. 5 extends the language to produce a tool definition using the approach; Sect. 6 discussed the approach and outlines further work.

2 Related Work

Software engineering is still essentially a young discipline and while efforts to compare it with other engineering disciplines invariably result in criticism of the discipline, some of the characteristics of software engineering are really the result of progress in trying to address the challenges of intense design. Young and Faulk provide an account of how this intense design manifests itself [22]. For example, unlike in other engineering areas, SE includes considerable activity in designing the design processes (see Pedreira et al. for relatively recent systematic literature review on tailoring software processes[18]). Closely related to the design process is the use of abstractions and notational elements. While abstraction is an essential step in any field of endeavour, in SE, the choice of abstraction is a defining characteristic of a software design process and can have direct and practical consequence on the output of a software design process as noted in the pioneering work of Parnas [17]. Similarly the use of notations to support abstractions is also a dominant field of enquiry. Thus programming languages, requirements descriptions [12], test plans [3] and so on have all been studied from a notational perspective.

Collectively, these activities have coalesced into ubiquitous approaches to software development manifested as methods, concepts and their associated software tools. There is no space to enumerate each and every one but examples include: The Information Engineering Facility from Texas Instruments [11], Software through Pictures [20] and of course the unification of modelling concepts, and notations through the Unified Modeling Language (UML) and its leading proponent for many years, Rational Rose and subsequent versions. As the use of UML grew, the complexity of the availability of concepts, notations, and the rules of their usage became much more of challenge and methods such as the Rational Unified Process developed alongside UML to help manage the complexity.

Despite the expressive power of UML, activity, consistent with SE research for designing the design processes, proceeded to extend UML or create variants of UML to support particular niche needs. For example, Egyed and Kruchten modified UML

and the Rational Rose toolset to support Architecture Modelling [7]. While such extensions are popular, they are technology oriented and do not present themselves at sufficient level of abstraction for end-users. As Kelly notes: "Simply put, UML is not domain-specific, and UML tools were not designed to support changing UML. Trying to build domain-specific models in a UML tool is—at best—like trying to write English in a Spanish version of Word" [13]. However, this style of modification that capitalized on the unique power of the conceptual structures when modifications are applied to the structures themselves remains a key strategic method in dealing with complex design requirements for producing software and so we can observe "abstractions of abstractions" and even the design of processes for design processes. Such a reflective approach or meta-engineering to modification creates a unique view of software engineering that is distinct to engineering in general as it is not constrained by material processes or laws of physics.

The idea of organizations designing processes to suit their software development practice and to have these supported by toolsets led to the development of meta-modelling toolsets. Although the original meta CASE (Computer aided Software Engineering) tools such as Ipsys Tool Builder [1], have long gone there are still two key toolsets widely used currently. These are MetaEdit+ [14] and GMF.[1] Egbert and Hainaut describe four necessary components that require modelisation in order to support the generation of tools from a meta-engineering environment [8]: (1) meta model (ontology and repository); (2) their interface (specification, representation and control); (3) their functions/processes (transformations and evolutions) and (4) their methodologies (the reasoning and activity guidelines). Both MetaEdit+ and GMF provide modelisation of these components. But it could be argued that the proprietary nature of the repository associated with MetaEdit+ prevents transformations to external tools and therefore sharing and evolution outside of the toolset. GMF and its dependence upon the Eclipse framework is simply too complex and so prevents users from designing domain-specific tools.

Tool frameworks emerged in recent years including Visual Studio and Eclipse. These frameworks allow the developer to tailor the platform to suit a particular domain by supplying static descriptions (often in XML) of the tool functionality and organization. However, they remain very platform specific, are incomplete, and require detailed implementation knowledge to work effectively.

A problem with tool descriptions is that they often use an implementation-specific or proprietary format. They often do not address semantics and where they aim to be self-contained, they present a mechanism for constructing a visual editor for a language such as MetaEdit+[2] and the Miarama toolset [9] (although see [4] that discusses semantic interoperability of DSMLs).

Language engineering can be argued to be a key feature of software system engineering and the underlying methods and technologies of meta-modelling [19] and domain-specific language engineering [21] have matured over the last decade to

[1] http://www.eclipse.org/modeling/gmf/.

[2] http://www.metacase.com.

the point where they can be applied in industrial situations. However, the application of language engineering to tool development has not seen significant research. This chapter represents a contribution to this field.

3 Language Engineering for Tools

Our proposal is that Software Engineering (SE) is a language-based discipline. The underlying design of software artefacts arising from an SE project should be located in the design of an appropriate language. Thus any SE project involves one or more domains that should be identified and precisely defined as languages. Furthermore, tools are required in order to engineer domain artefacts and tool definition should be part of the domain engineering process. Once a precise requirement for a DSL tool is available, a project is in a position to find the most appropriate way of providing the tools and their associated languages.

This approach can lead to a specification for a project-specific tool suite that supports domain-specific languages. As such it represents a lens through which all concrete technologies can be viewed in order to achieve consistency and clarity across a project. Taken further, the approach can lead to executable models in which case, given the availability of a suitable meta-tool platform, the project-specific tool suite can be automatically generated.

This chapter addresses the specification of languages and tools, and this section describes an approach for engineering these domains. Section 3.1 describes potential approaches to language description, Sect. 3.2 shows how one approach can be implemented using standard modelling techniques and Sect. 3.3 shows how the approach incorporates tooling.

3.1 Domains as Languages

Languages are the medium of communication between humans. Geographical distance can explain the difference between Chinese and English, but other than that we might initially be tempted to conclude that languages in geographical areas are essentially the same. However, there is a rich diversity of special purpose languages that have arisen, some of which cross-cut national boundaries, in order to support meaningful communication in specialized areas, or *domains*. Most professional disciplines constitute such domains, for example medicine (try interpreting your doctor's notes!) and programming (where bugs cannot fly but need to be exterminated). The importance of language and its relevance to computer system and organizational interaction has been researched at depth by foundational work [5,15].

Computer systems are controlled by making demands on their behaviour and requesting information about their state. The particular demands and requests for any application constitute a language of discourse with the system. Like languages

associated with nation states, there is a general purpose language for controlling families of systems; in the early days of computing this was the mode of discourse. However, like discourse in professional domains, such low-level communication was seen to be error prone and verbose. This led to the development of high-level languages.

Software engineering differs from other engineering disciplines, such as Civil, Electrical and Chemical that are each based on a collection of fixed rules that are the same for each new system. Software is capable of defining new execution rules for each new system or family of systems through programming. The dichotomy of program and data is not fixed and allows programs to be represented as data, and data to be interpreted by programs. This recursive relationship can be extended as far as required in order to shield the engineer from the low-level communication medium. Therefore, what works for human-to-human communication (i.e., the development of domain specific languages of discourse) can be applied to software systems with the same benefits.

Given that software engineering can be viewed as a language-based activity, how frequently should a language be developed and who should do it? Once for all applications by hardware specialists? Once for each operating system? Once for each development style (OO, logic)? Once for each type of application (real-time, graphical, distributed)? Once for each application (booking system, share transaction system)? Once per use by users? Clearly there is a spectrum and it will depend on the range of variability required as to where the requirement to engineer and use languages lies. In general, one could argue that a general application user would not require the ability to specialize the medium of communication, although many systems allow a degree of control over system properties that can be viewed in this light.

Commercial software development is increasingly striving for application families, or *product lines*. Such an approach is attractive because it can abstract key commonalities and isolate variation points. In order to succeed with this approach it is necessary to be clear about what constitutes a language and what range of variability is available to the engineer. A *language* for software engineering must have a single definition that we will call a model of its *abstract syntax*. In software terms, the abstract syntax is a computer-friendly data representation of the language and provides the reference-point for all other definitions of the language; the exact format of the data definition is a variation point for language definition. Typically, humans find abstract syntax difficult to work with because it is verbose, therefore a language has one or more *concrete syntax* definitions. A concrete syntax for a language may take the form of strings of text or of graphical displays, or both [2]; it provides a variation point and is often a matter of taste. A concrete syntax definition may take the form of a grammar and there will be a relationship between the concrete and abstract syntaxes.

A language should have a *semantics*. A semantics is typically defined as a model of a *semantic domain* (separate from the abstract syntax model) and a relationship between the abstract syntax and the semantic domain. Degenerately, there is nothing special about the semantics of a language, it is composed from two models and a relationship and is defined for a particular purpose. A semantics may be defined

in order to deduce properties of language elements, or to provide a mechanism for defining valid and invalid abstract syntax configurations. Often when people talk of *the* semantics of a language, they really mean one of the possible relationships that can usefully be defined.

Computer Science has defined a number of categories of language semantics [16], including:

Denotational. Where each language element is to be viewed as directly representing a semantic domain element. In this case the semantic definition takes the form of a predicate that holds between elements from each domain. For example, a language for class modelling has models that denote sets of objects and links. The objects are required to be well-formed instances of classes in the model and links are instances of associations that must hold between appropriate objects and must satisfy multiplicity constraints attached to the association ends.

Operational. Where each language element is to be viewed as representing a sequence of steps in a calculation. In this case the semantic definition can take the form of an algorithm that performs the steps, or the form of a relationship between a syntax element and a state sequence. For example, an interpreter can be defined that processes a state machine and a collection of externally generated events in order to control an object.

Axiomatic. Where each language element takes part in a relationship with semantic domain elements that represents facts that can be deduced and the semantics takes the form of a theory. For example, a class model may include operations with pre and post conditions written in OCL. From such a model, properties of legal execution traces can be deduced.

The particular semantic category that is chosen will depend on the type of language and its intended use. Two obvious cases exist in software engineering which make a distinction between *static* and *dynamic* systems. Typically a static system represents some information, such as a relational database or an XML document, expressed using a convenient domain-specific language. A dynamic system executes in some way, for example a business process that manages a collection of databases. Static systems lend themselves to denotational semantics and dynamic systems lend themselves to operational semantics. Both static and dynamic systems can be expressed using axiomatic semantics.

3.2 A Language Based Method

The previous section has presented the case for languages-based software engineering and has outlined the components necessary to define these kinds of languages. To use this approach on a particular project, a specific technology and language definition method must be selected. There are many technologies that are suitable for defining languages including BNF grammars, symbolic programming languages, DSL tools, graph grammars, term rewriting systems. For the purposes of this chapter

we will use UML class diagrams together with invariant constraints expressed using OCL since these have been standardized and are widely used. In addition we will limit ourselves to static languages and denotational semantics.

UML is to be used to *specify* DSL tools. Therefore models are used to express the language features described in the previous section and predicates are used to define relationships between the language features. As such the models are required to express *what* needs to hold between the models, even when the models describe dynamic features such as the actions performed by a tool when a user selects a menu item. This is to be contrasted with DSL models that express *how* the relationships are constructed; such models would be executable and are not considered further.

A specification in UML can be expressed using *relationship* models. A relationship model consists of a collection of classes that hold between elements of two or more *domain* models. The domain models are to be viewed as being imported into the relationship model. Each class in the relationship model is linked to domain model classes using associations and OCL invariants on the relationship classes are used to specify when the relationships hold between domain instances. The choice of UML leads to the following:

Abstract syntax. A domain model showing the concepts and relationships in the language. OCL is used to define syntactic constraints between elements. A typical syntactic constraint would require all names to be unique in a given context.

Concrete syntax. A domain model for a collection of display elements. Typical modelling examples would be a graphical display language consisting of boxes, text and lines. In principle there may be more than one concrete syntax model. The case study in Sect. 4 uses graphical elements such as box and text because the language lends itself to a diagrammatic syntax. A general purpose concrete syntax domain may be extended, for example class-boxes can be defined as a collection of sub-boxes and text. Well-formedness constraints are expressed using OCL.

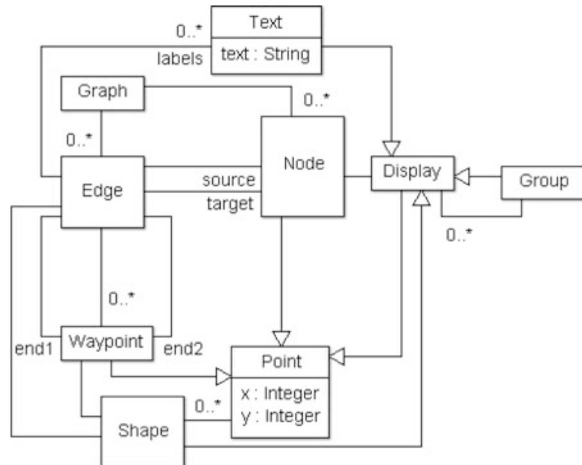
Syntax relationship. A relationship model between abstract and concrete syntax elements. This model defines the constraints on the legal concrete representations of well-formed language constructs.

Semantic domain. A domain model whose elements are to be used as the meaning of abstract syntax elements. It is not necessary to have the abstract syntax and semantics elements to be in one-to-one correspondence. OCL is used to express well-formedness constraints on configurations of semantic elements.

Semantic relationship. A relationship model between abstract syntax and semantic domain. It is important to note that there may be more than one semantic domain and therefore more than one semantic relationship. However, in practice it is usual to have a single semantics and therefore the relationship model can be viewed as defining *the* semantics of the language.

A language is rarely defined from scratch; it is usually based on existing languages. If it is a textual language, then there is generally a fairly standard expression sub-language, and if it is a graphical language, then it tends to be based

Fig. 1 General concrete syntax



on a graph or a tree. The language used in our case study is graphical and the concrete syntax will be based on the general purpose graphical syntax defined in Fig. 1. A diagram is a graph consisting of nodes with edges between them. Each node occurs at a point on the diagram and has a display element that is used to draw the node on the screen. A display element can be some text, a shape made up of points with lines between them, or is a group of displays. Each edge has a source and target node, and a collection of waypoints. An edge has a minimum of two waypoints that are the edge-ends. An edge has a collection of labels and each waypoint of an edge has an option shape used to decorate the waypoint.

3.3 The Tooling Domain

Our proposal is that tooling should be included in any domain-specific approach to software engineering. By providing a specification of the required tooling for a language, a project documents the intended usage of a DSL and clearly expresses the requirements for any concrete tools that are to be used. Where multiple languages are to be used, it is possible to test concrete tool frameworks against the combined tool requirement model.

Given this proposal, how are tools to be specified? This can be determined by analysis of the key features of any tool. A tool exists in a collection of states that can be changed by interaction with its environment; this is equivalent to the abstract syntax of the language as defined above. Since the tool must manage instances of a language, its states are linked to abstract syntax instances.

Most software tools have a user-interface that consists of tree browsers, text editors, property editors, graphical editors etc. The user-interface is equivalent to the concrete syntax of a language as described above and the same approach of a relationship model between abstract and concrete syntaxes can be used.

A tool performs tasks, often in response to user interaction through a collection of input gestures via buttons, menus etc. Tool execution can be modelled as a collection of state-transition traces, where each step in a trace contains before and after states, and an event. Such executions constitute the semantics of a tool, defined in terms of a semantic domain and a relationship model.

Therefore, a tool can be modelled in terms of the same features as a language as described in the previous section. If the tool is to be used to manage a language, then the two models can be linked via relationship models so that the language specification restricts legal tool behaviour and the tool restricts the features of the language that are available to the user.

This approach is attractive because it places no restrictions on how the tool is realized. The behaviour can be left ambiguous where further detail is not required. A typical ambiguity is diagram layout where any concrete tool would be free to use any algorithm. Multiple tools that are specified in this way can be combined using relationship models to produce a single unified tool model; each component tool will place restrictions on the functionality and behaviour of the other.

4 Case Study

The previous section has argued the case for a language-based approach to tool-based domain engineering. This section provides an overview of applying this approach to a domain. We have chosen a domain that is widely understood in order that the key steps of the process are clear. The case study uses class modelling as the domain; the following sections address each major step in the approach: the abstract syntax of class-based modelling is defined in Sect. 4.1; the language for *drawing* class models is defined in Sect. 4.2; Sect. 4.3 defines a relationship between the elements of the class models and the concrete syntax, the relationship specifies when a drawing is a legal representation of a model. For the case study we are taking a denotational approach to language engineering and therefore Sect. 4.4 defines a model of the semantic domain for class models: every class model denotes a snapshot contain objects and links. Given an abstract syntax and a semantic domain it is necessary to define when a semantics (i.e., a snapshot) is a well-formed instance of a class model, this is done by defining a relationship between elements of these two models in Sect. 4.5.

Note that this section aims to give the key features of all elements of the approach. As such some of the models omit elements that would otherwise be necessary to provide a complete definition where the inclusion of these elements simply repeats key features that are exemplified elsewhere. In addition it should be noted that association ends on models are labelled only when necessary and that the names of association ends default to the names of the attached classes with a lowercase initial letter and internalized capitalization. Multiplicities on association ends default to 1 unless otherwise indicated.

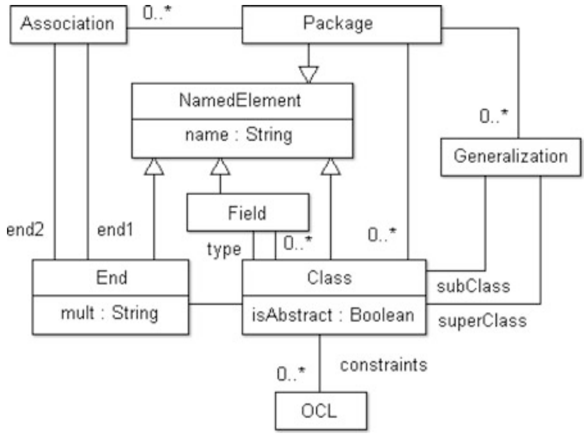


Fig. 2 Class models abstract syntax

4.1 *Abstract Syntax Domain Model*

The abstract syntax structure of class models is shown in Fig. 2. A package contains a collection of classes that can be related using associations and generalizations. Each class has a number of fields and a collection of constraints. Each constraint is an OCL expression; it is beyond the scope of this chapter to define OCL abstract syntax; however, the reader is assumed to be familiar with standard first-order predicate calculus which OCL approximates. An association has two ends attached to classes, each end has a name and a multiplicity. The syntax for class models has been chosen so that it is the basis for the languages used in the case study. A full and complete model for class modelling is of course available in the UML 2.x specification available at: http://www.omg.org/spec/UML/2.0/. Our presentation is necessarily a simpler version for the purposes of illustrating key concepts of our approach.

A domain has a collection of well-formedness constraints. it will not be possible with the scope of the chapter to define all of the well-formedness constraints for class diagram domain and relationship models. Therefore, we will give a representative sample. For example, the following constraints requires that all classes in a package have different names:

```
context Package inv:
  classes->forAll(c1 c2 | c1.name = c2.name implies c1 = c2)
```

4.2 *Concrete Syntax*

In general, the concrete syntax of a language will be built from some basic elements. The concrete syntax of class diagrams is defined as an extension to the domain model given in Fig. 1. The extensions specialize the basic concrete elements so that it is easy to define the mapping model between the abstract and concrete syntaxes.

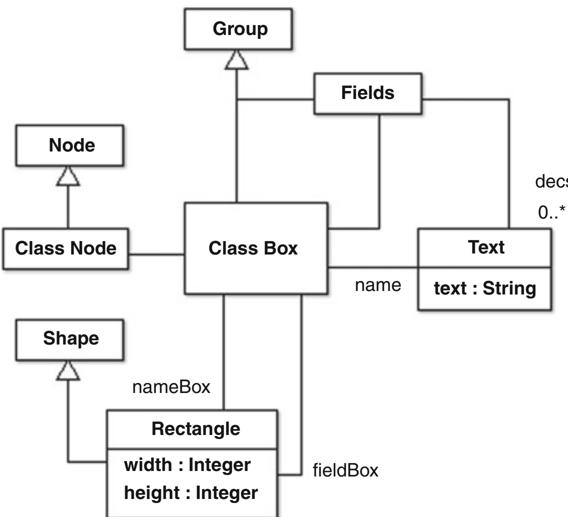


Fig. 3 Class nodes

The specializations for classes are shown in Fig. 3 where the class `ClassNode` is a specialization of `Node` that represents classes. Classes are displayed as rectangles containing text, so `Shape` is specialized to produce `Rectangle`. `ClassBox` is a specialization of `Group` that is required to contain the display elements for a class name and its fields. OCL is used to require the elements to be correctly formed, for example the following is a fragment that forces the nested rectangles in a class box to be positioned correctly:

```
context ClassBox inv:
  displays = Set{nameBox,fieldBox,name,fields} and
  nameBox.x = x and nameBox.y = y and
  fieldBox.x = x and
  fieldBox.y = y + nameBox.height and
  nameBox.width = nameBox.width
```

The domain model in Fig. 4 defines a specialization of edges that can be used to represent associations and generalizations on a diagram. As before OCL can be used to constrain the domain, for example the type and positioning of association edge connections (where `contains` holds between a display and a point when the display contains the point):

```
context AssociationEdge inv:
  source.oclIsKindOf(ClassNode) and
  target.oclIsKindOf(ClassNode) and
  source.display.contains(end1) and
  source.display.contains(end2)
```

4.3 Syntax Mapping

A syntax mapping is a model that links elements from the abstract syntax domain to appropriate elements in the concrete syntax domain. In general the mapping

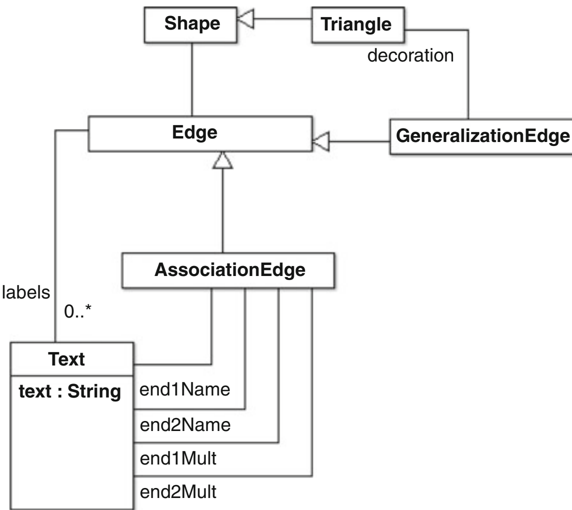


Fig. 4 Class model edges

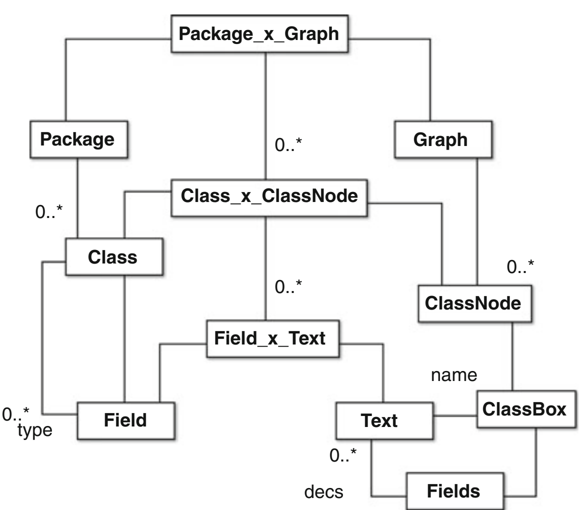


Fig. 5 Class syntax mapping

describes the minimum constraints necessary to ensure that the rules for constructing and displaying elements of the language are correct. Typically the syntax mapping will leave issues such as layout underspecified so that tools can implement their own algorithms. In addition, a syntax mapping may leave features such as colour and the use of icons unspecified where these are not important.

Figure 5 shows the mapping model for classes within packages. It shows a typical pattern that occurs in mapping models between two domains `X` and `Y` whereby mapping classes `A_x_B` are constructed for class `A` of domain `X` and class `B` of domain `Y`.

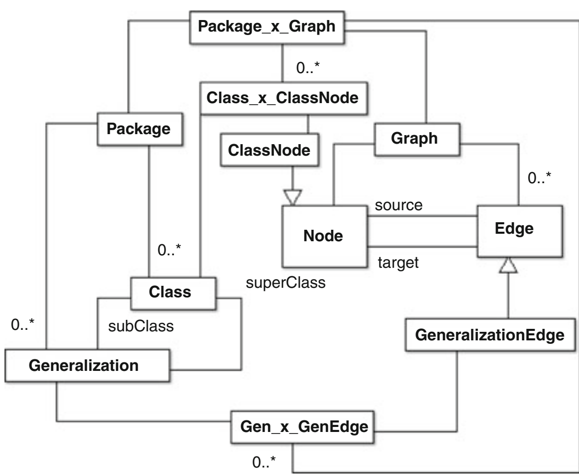


Fig. 6 Generalization syntax mapping

The root mapping class is `Package_x_Graph` that defines a relationship between a package and a graph. The OCL constraints attached to a mapping class define the conditions under which instances of the associated domain instances can be related. For example, a graph is a correctly formed package when it has a class node for each class in the package:

```
context Package_x_ClassNode inv:
  package.classes = classNode_x_classes.class and
  graph.nodes = classNode_x_classNodes.node
```

Each mapping class must have appropriate constraints. For example, the name of a class must be associated with the text in the class node of a class node:

```
context Class_x_ClassNode inv:
  class.name = classNode.classBox.name.text and
  class.fields = field_x_texts.field and
  classNode.classBox.fields.decs = field_x_texts.text
context Field_x_FieldText inv:
  text.text = field.name + ":" + field.type.name
```

The mapping model for generalizations between classes is shown in Fig. 6. The mapping constraints are not defined here, but are of a similar form to those defined for `Package_x_Graph`.

4.4 Semantic Domain

The semantic domain of a language defines the meaning of the elements from the abstract syntax domain. Class models defined in Fig. 2 are static since there is no way of defining dynamic features such as operations. Class models are well understood and therefore the semantic domain shown in Fig. 7 is perhaps obvious. However, when working with a new domain, the semantics might be less obvious. In these situations it is worth considering designing the semantic domain before the

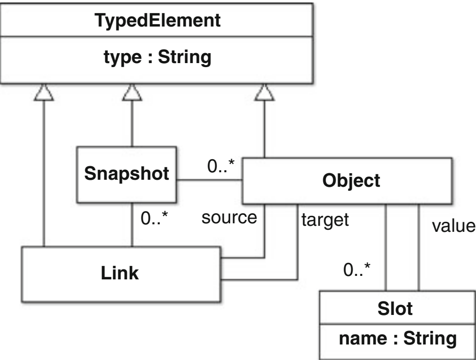


Fig. 7 Class semantic domain

syntax domains. This can be done by constructing a model of the things that are to be denoted and then designing the syntax domains in such a way as to provide a convenient way of denoting the semantic domain elements.

As with the syntax domains, a semantic domain includes OCL constraints that express well-formedness constraints. For example, snapshot constraints should include a requirement that object fields must have unique names and links can only hold between objects that are in the same snapshot.

4.5 Semantic Mapping

The semantic mapping associates abstract syntax elements with semantic domain elements. In the case of packages, classes are associated with objects so that the slots and fields match up as shown in Fig. 8. The following OCL constraint requires that packages have snapshots as instances when the objects in the snapshot are all instances of corresponding classes in the package:

```
context Package_x_Snapshot inv:
  package.classes->includeAll(snapshot.objects) and
  package.classes = class_x_objects.class and
  snapshot.objects = class_x_objects.object
```

Each instance of a class must have slots that correspond to the fields of the class:

```
context Class_x_Object inv:
  class.fields = field_x_slots.field and
  object.slots = field_x_slots.slot
```

The names of fields and slots must match up:

```
context Field_x_Slot inv:
  field.name = slot.name
```

The type of a field must correspond to the type of a slot value:

```
context Package_x_ClassNode inv:
  class_x_objects.field__x_slots->forAll(r1 |
    class_x_objects->exists(r2 |
      r1.class = r1.field.type and
      r1.object = r1.slot.value))
```

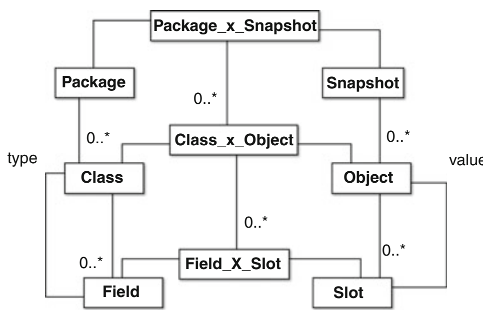


Fig. 8 Instance semantics

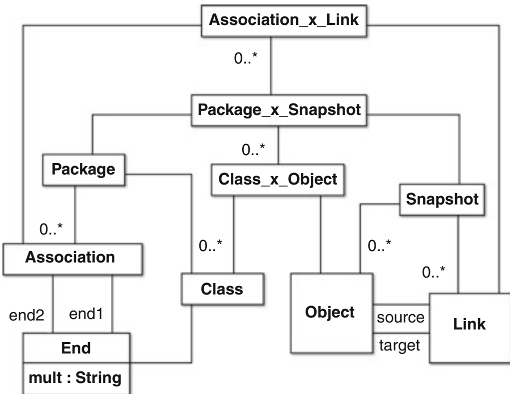


Fig. 9 Association semantics

If we assume that each OCL expression has a predicate `satisfiedBy` that returns a boolean value when supplied with an object then the invariants on classes are specified as:

```
context Class_x_Object inv:
  class.constraints->forAll(c | c.satisfiedBy(object))
```

The semantic mapping for associations and links is defined in Fig. 9. The multiplicity on association ends places a constraint on the number of links that can occur in the snapshot. For example, suppose that `A` is an association between classes `X` and `Y` with the multiplicity 1 on the end (`end1`) attached to `X`. In a snapshot we equate a link `source` with `end1` and a link `target` with `end2`. Therefore, for a link to be a valid instance of `A`, the source must be a valid instance of `X`, the target must be a valid instance of `Y` and, for each instance of `Y` there can be at most one instance of `A`. The multiplicity constraint is expressed in OCL as follows:

```
context Package_x_Snapshot inv:
  package.associations->forAll(a |
    a.end1.mult = "1" implies
      snapshot.objects->forAll(o |
        association_x_links->select(r |
          r.association = a and
          r.link.target = o)->size >= 1
```

We omit the constraints relating to generalizations that require an object to have slots for all the super-classes of a class.

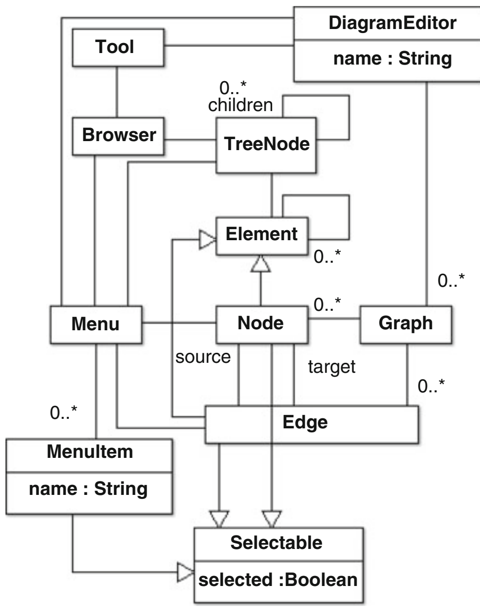


Fig. 10 Tool abstract syntax

5 Tooling

Software tools are a huge investment in terms of development and maintenance. Many tools tend to be large and complex to learn. The ability to specify the requirements of a tool for a specific task makes it clear how any suitable tool should be used and also makes the required functionality independent of any specific technology platform. Once defined, a tool can be reused by transforming its specification to a new language. This section provides an outline on how a tool can be defined for the class modelling domain by using a language-based approach.

5.1 Abstract Syntax

Figure 10 defines an abstract syntax domain for a simple general-purpose modelling tool. The tool consists of a browser and a diagram editor. The browser contains tree structured data and the diagram editor contains a graph. Various elements of the tool have menus associated with them.

Note that the tool model is abstract in the sense that it does not specify the format of the tree structured data or the graph data. Therefore, OCL can be used to place some requirements on the tool model, for example that menu items are all unique in a menu, that will apply to any tool that is specified to be consistent with the model via a mapping model.

5.2 Concrete Syntax and Syntax Mapping

The concrete syntax of a tool is defined using a model of the required graphical elements to be used for browser nodes, menus and diagrams. This can simply use the class `Display` defined earlier:

The abstract and concrete syntax of a tool are combined using a mapping model:

The mapping class `Tool_x_Window` has constraints that require the window to include suitable arranged display elements. For example, the tree nodes of a browser node must be displayed as text elements with positions that are arranged as a tree, and the edges of a diagram must be displayed as lines whose end positions match the locations of the attached nodes. It is beyond the scope of the current chapter to give the details of the `Tool_x_Window` mapping class; however, it should be noted that the approach allows the mapping to be underspecified with respect to the exact display elements used for tool features such as menus, diagram nodes and waypoint decorations. In addition, the mapping may also be used to allow certain usability features of the required tools to be specified. For example, if colours are available in the `Window` concrete syntax model, then certain mixtures of colours can be defined as illegal.

5.3 Semantics

In general, a software tool is a reactive system that performs actions in response to a stimulus provided by the user. Each action causes the tool to change state and then wait for the next stimulus. Figure 11 shows a simple semantic domain for a tool as a *filmstrip* consisting of ordered steps that perform tool actions. A simple tool for class modelling will require actions that create, delete and move elements. More sophisticated tools will include actions for drag-and-drop, save and load, user input etc.

5.4 Semantic Mapping

Our proposition is that a tool can be treated as a domain-specific language and therefore should be specified using a semantic mapping. The semantic mapping

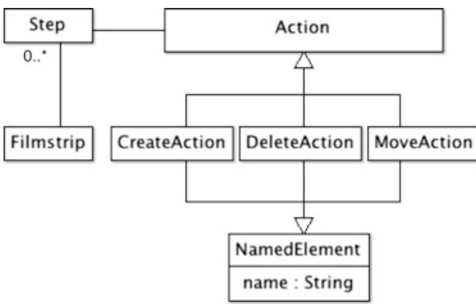


Fig. 11 Tool semantics

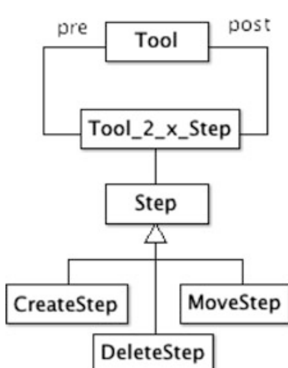


Fig. 12 Tool step semantic mapping

will associate elements of the tool abstract syntax with the tool semantic domain. In the case of simple class modelling, the semantic mapping must define what happens for the creation, deletion and movement actions. Figure 12 defines a mapping model consisting of a pre and post tool state and a step. The constraints attached to the mapping class must be carefully constructed in order to specify the required state changes, although some of the details may be left under-specified in order to allow tools the greatest amount of freedom in satisfying the specification. For example, the following constraint requires that a browser node is created in response to a create action. It assumes that a tool provides the query operation `hasSelectedBrowserNode()` which is true when exactly one browser node is selected and `getSelectedBrowserNode()` that gets the selected node. In addition we assume that we can perform `p - q` for two tools `p` and `q` that will produce the elements in `p` that are not elements in `q`.

```
context Tool2_x_CreateStep inv:
 pre.hasSelectedBrowserNode() and
 post.hasSlectedBrowserNode() and
 let n = pre.getSelectedbrowserNode()
     n' = post.getSelectedBrowserNode()
 in n.menuItems->exists(i | i.name = step.action.name) and
    (post - pre) = n'.children - n.children and
    n'.children->size = n.children->size + 1
```

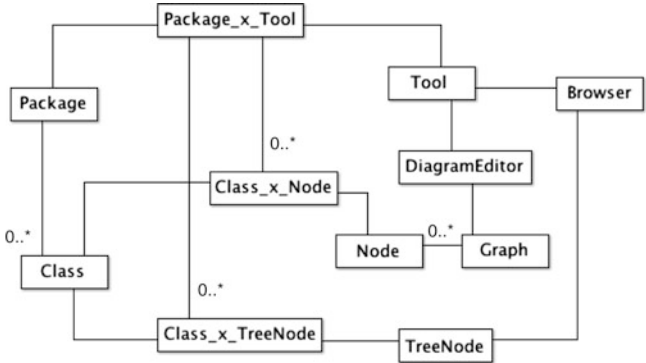


Fig. 13 Mapping abstract syntax

5.5 Mapping Languages and Tools

A tool is used to manage elements in a language. Therefore, the abstract syntax of the language must be associated with the abstract syntax of the tool. In the case of class modelling, packages and package elements are associated with the appropriate tool elements such as browser nodes and diagram nodes. Figure 13 shows a mapping model that defines the appropriate associations for classes (associations and generalizations are omitted). OCL constraints are used to tie the elements together, for example:

```
context Package_x_Tool inv:
  package.classes = class_x_nodes.class and
  tool.nodes = class_x_nodes.node and
  package.classes = class_x_treeNodes.class and
  tool.browser.topLevelTreeNodes() = class_x_treeNodes.treeNode
```

Having linked the abstract syntax of the tool and class modelling language, the concrete syntax of both languages must be mapped onto each other. Although verbose, this is a straightforward mapping model that involves point-wise constraints between the display elements used to construct class models and the display elements that are drawn on a window. The mapping class `P_x_G_x_T_x_W` associates the mappings `Package_x_Graph` and `Tool_x_Window`:

the following OCL constraint shows how the two mappings are associated:

```
context P_x_G_x_T_x_W inv:
  t_x_w.displays->includesAll(p_x_g.graph.nodes.display) and
  t_x_w.displays->includesAll(p_x_g.graph.edges.shapes) and
  t_x_w.displays->includesAll(p_x_g.graph.edges.waypoints.shape) and
  t_x_w.displays->includesAll(p_x_g.graph.edges.waypoints.labels) and
  t_x_w.displays->select(t | t.oclIsKindOf(Text))->
    includesAll(p_x_g.package.classes.name) and
```

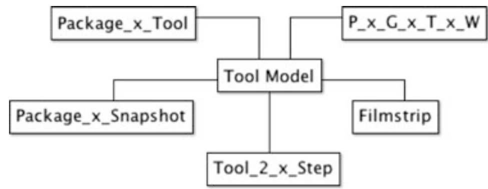


Fig. 14 The complete tool specification

```
t_x_w.displays->select(t | t.oclIsKindOf(Text))->
  includesAll(p_x_g.package.associations.name) and
t_x_w.displays->select(t | t.oclIsKindOf(Text))->
  includesAll(p_x_g.package.associations.end1.name) and
t_x_w.displays->select(t | t.oclIsKindOf(Text))->
  includesAll(p_x_g.package.associations.end2.name)
```

The mapping defined above requires the tool window to contain all the appropriate display elements but does not place any restrictions on where the information is displayed. The tool syntactic mappings defined above require that the information is appropriately displayed via browser areas and diagram areas.

The tool for the case study in this chapter has been defined using a language-based approach in terms of its abstract syntax, concrete syntax and its semantic mapping. As such it is sufficiently underspecified and may be used to specify a tool for any language that involves a browser and diagram elements. We have then defined point-wise mappings between class model syntax and tool syntax and between class model semantics and tool semantics. It remains to ensure that all of the point-wise mappings hold simultaneously, thereby completing the tool specification. Figure 14 shows the mapping `Tool Model` that maps all the component mappings. The associated OCL constraint (not shown) on `Tool Model` requires that the appropriate elements of each constituent mapping tie up correctly. Therefore, both syntactically and semantically the tool will behave as required for all steps in a filmstrip.

6 Analysis and Conclusion

This chapter has identified a problem with software engineering tools and modelling tools in particular that the investment required to produce the tools leads to large complex general purpose platforms that are expensive to use and difficult to learn. A solution to this problem is to take a domain-specific approach to tooling, thereby producing lean focussed tools that are appropriate to each new project. Whilst domain-specific tools solve some problems, they introduce others since they are often based on proprietary technologies that lead to interoperability issues.

Our proposal is to take a domain engineering approach to tooling so that the tools required for a project are specified as models and are then mapped on to existing general purpose technologies or on to domain-specific technologies as appropriate. We have described a method for specifying domain-specific tools based on UML and used the method to specify a simple tool for class modelling.

The benefits of domain-specific tool models include a precise description of tools and the languages they support. The models can be mapped to the functionality of existing platforms and the tool semantics can be used to check that the platforms satisfy the required semantics. In addition tool models can be reused. For example, the class modelling tool defined in this chapter can be extended to include features such as components and state machines. Furthermore, the tool models can be sliced to restrict their functionality, for example a class modelling tool that does not support generalization. Since the tool model contains both the language and its tool functionality, any extension or restriction of the language must also indicate the extra tool functionality or the tool functionality that can be hidden from the user.

The method and case study in this chapter has shown that it is possible to specify a tool in terms of a domain and mapping models. The case study that has been used is an example of a *horizontal* domain and it remains to show that the approach works for a variety of vertical domains. In addition the horizontal domain, although familiar, is self-referential in the sense that a class-modelling tool has been specified using class models. It remains to apply the approach to another horizontal domain that is not used in the definition of itself.

Our goal is to be able to construct executable models of tools and to provide a standard meta-language for tools that can be consumed by a wide range of tooling platforms. A number of approaches to this problem have been tried, but most do not explicitly model all of the features of a model. A successful approach, that unfortunately uses a non-standard representation but otherwise shows that the approach could be generally applied, is the meta-modelling tool platform XMF that is compared against similar software tools in [10].

References

1. Alderson, A.: Meta-case technology. In: Software Development Environments and CASE Technology. Lecture Notes in Computer Science, vol. 509, pp. 81–91. Springer, Berlin (1991)
2. Andrés, F., de Lara, J., Guerra, E.: Domain specific languages with graphical and textual views. In: AGTIVE, pp. 82–97, 2007
3. Bauer, J.A., Finger, A.B: Test plan generation using formal grammars. In: Proceedings of the 4th International Conference on Software Engineering (ICSE'79), pp. 425–432. IEEE, Piscataway (1979)
4. Chiprianov, V., Kermarrec, Y., Rouvrais, S.: Meta-tools for software language engineering: a flexible collaborative modeling language for efficient telecommunications service design. In: FlexiTools: Workshop on Flexible Modeling Tools at the 32nd ACM/IEEE ICSE Intl. Conf. on Software Engineering, 2010
5. Clancey, W.J.: Understanding computers and cognition: A new foundation for design. Artif. Intell. **31**(2), 232–250 (1987)
6. Clark, T., Bettin, J.: Editorial for the theme issue on model-based interoperability. Software Syst. Model. **11**(1), 7–10 (2012)
7. Egyed, A., Kruchten, P.B. Rose/architect: a tool to visualize architecture. In Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences (HICSS'99), vol. 8, p. 8066. IEEE Computer Society, Washington, DC (1999)

8. Englebert, V., Hainaut, J.L.: Db-main: a next generation meta-case. Inform. Syst. **24**(2), 99–112 (1999)
9. Grundy, J.C., Hosking, J.G., Huh, J., Na-Liu Li, K.: Marama: an eclipse meta-toolset for generating multi-view environments. In: ICSE, pp. 819–822, 2008
10. Helsen, S., Ryman, A.G., Spinellis, D.: Where's my jetpack? IEEE Software **25**(5), 18–21 (2008)
11. Texas Instruments Incorporated: IEF: Methodology Overview, 2nd edn. Texas Instruments, Plano (1990)
12. Jacobson, I.: Object-oriented software engineering: a use case driven approach. Pearson Education India, 1992
13. Kelly, S.: Comparison of eclipse EMF/GEF and metaedit+ for DSM. In: 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Best Practices for Model Driven Software Development, 2004
14. Kelly, S., Lyytinen, K., Rossi, M.: Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In: Advanced Information Systems Engineering, pp. 1–21. Springer, New York (1996)
15. Kensing, F., Winograd, T.: The language/action approach to the design of computer-support for cooperative work: A preliminary study in work mapping, Number 27. Stanford University. Center for the Study of Language and Information, 1991
16. Milne, R., Strachey, C.: A Theory of Programming Language Semantics, 99th edn. Halsted Press, New York (1977)
17. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Comm. ACM **15**(12), 1053–1058 (1972)
18. Pedreira, O., Piattini, M., Luaces, M.R., Brisaboa, N.R.: A systematic review of software process tailoring. ACM SIGSOFT Software Eng. Note **32**(3), 1–6 (2007)
19. Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G.: Metamodelling - state of the art and research challenges. In: Model-Based Engineering of Embedded Real-Time Systems, pp. 57–76, 2007
20. Wasserman, A.I., Pircher, P.A.: A graphical, extensible integrated environment for software development. In: ACM Sigplan Notices, vol. 22, pp. 131–142. ACM, New York (1987)
21. Weisemöller, I., Schürr, A.: A comparison of standard compliant ways to define domain specific languages. In: MoDELS Workshops, pp. 47–58, 2007
22. Young, M., Faulk, S.: Sharing what we know about software engineering. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, pp. 439–442. ACM, New York (2010)

Modeling a Model Transformation Language

Eugene Syriani, Jeff Gray, and Hans Vangheluwe

Abstract Domain-specific modeling techniques can reduce the gap between the problem space and the solution space by using abstractions and notations that represent domain concepts. The fact that only familiar concepts and notations are used in the model allows domain experts to understand and be involved directly in design. The resulting artifacts of this process are models and transformations. There are well-known techniques for developing modeling languages (e.g., metamodeling and synthesis of modeling environments); however, there is currently no well-defined technique for engineering model transformation languages (MTLs). This chapter introduces a language engineering technique for building MTLs that is based on treating each MTL as a domain-specific language, more specifically, as languages for describing specific classes of transformations. In this approach, all the components of an MTL are modeled explicitly at the proper level of abstraction using the most appropriate formalisms. Consequently, this facilitates the automatic synthesis of MTL development environments and supports the evolution of model transformations, which assists domain experts in designing models and transformations in an integrated and uniform manner.

Keywords Domain-specific languages • Higher-order transformation • Language engineering • Model transformation

E. Syriani (✉) · J. Gray
University of Alabama, Tuscaloosa AL, USA
e-mail: esyriani@cs.ua.edu; gray@cs.ua.edu

H. Vangheluwe
Antwerp University, Belgium and McGill University, Canada
e-mail: hv@cs.mcgill.ca

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_9, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

Model-driven engineering (MDE) [34] is considered a well-established software development approach that uses *abstraction* to bridge the gap between the problem domain and the software implementation. The MDE approach supports systematic transformations of problem-level abstractions into their implementations. To bridge the gap between the application domain and the solution domain, MDE uses models to describe complex systems at multiple levels of abstraction, as well as automated support for transforming and analyzing models. This separation allows the description of key intellectual assets in a way that is not coupled to specific programming languages or target platforms.

MDE considers models and transformations as first-class entities. In MDE parlance, a *model* represents an abstraction of a real system, capturing some of its essential properties, to reduce accidental complexity present in the technical space. A model conforms to a *meta-model* [22], which defines the abstract syntax and static semantics of a modeling language, a (possibly infinite) set of models. A meta-model specifies the permissible syntax of a modeling language, often in the form of constraints. The developer can then manipulate models by means of *model transformation* [33]. Transformations allow one to define the dynamic semantics, execute, analyze, synthesize code, optimize, compose, synchronize, and evolve models. Model transformations are at the very heart of MDE [33].

There are well-known techniques for developing modeling languages. A popular approach uses meta-modeling [22] and synthesis of modeling environments [26] from the meta-model. Meta-modeling environments, such as the GME [27], MetaEdit+ [18], and AToM^3 [25], provide a language-development capability where the syntax and static semantics of the language are defined. Another popular approach for defining modeling languages is through extension, such as UML profiles [10]. In this chapter, we focus on graphical domain-specific languages (versus textual). There is currently no well-defined technique for engineering model transformation languages (MTLs). This chapter proposes to model both the syntax and semantics of model transformation languages explicitly using well-known modeling principles.

The following subsections briefly introduce modeling and transformations. In Sect. 2, we describe how to model an MTL at the syntactic level (abstract and concrete). In Sect. 3, the semantics of such transformation models are also modeled through the use of meta-modeling and model transformation. The aim is to increase the developer's productivity, by raising the level of abstraction at which transformations can be specified and by lowering the mismatch between MTLs and their application domain, i.e., minimizing accidental complexity. We illustrate the applicability of the proposed approach by re-designing a recent MTL, MoTif [38], following the MPM principles (defined in the next sub-section). Section 4 addresses the deployment and utilization of the re-designed MoTif. Section 5 discusses related work. We conclude in Sect. 6.

1.1 Modeling Principles

Domain-specific modeling (DSM) [11] is a branch of MDE that allows models to be manipulated at the level of abstraction of the application domain the model is intended for, rather than at the level of computing. In DSM, domain experts can create models that described some computational need using abstractions and notations that match their own domain of expertise. Thus, end-users who do not possess the skills needed to write computer programs using traditional languages (like Java or C++) can describe a task in a more familiar language.

Another fundamental pillar of MDE is *Multi-Paradigm Modeling* (MPM) [29]. MPM addresses complexity by explicitly modeling all aspects and parts of a problem, in all phases of the development process. Models cover different levels of abstraction (and the abstraction/refinement relationships between them) and may combine models in different formalisms. The explicit modeling includes model transformations and relationships as well as the development process. MPM promotes modeling all parts of the system, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s), to reduce accidental complexity. One key aspect of MPM is multi-abstraction. A model abstraction is a view of a system exhibiting some of its properties while hiding others. Multi-abstraction is thus the ability to express models at different levels of abstraction. MPM realizes that systems can be represented in different modeling languages or formalisms. MPM, in particular multi-abstraction and multi-formalism modeling, is enabled by the use of meta-modeling and model transformation. Instead of describing system behavior in terms of code, MPM principles state that transformations also should be modeled explicitly.

1.2 Background on Model Transformation

A model transformation receives as input a source model and transforms it into a target model, where both models conform to their respective meta-model. The meta-model for the source and target may have the same meta-model, which leads to an endogenous transformation; or, the meta-model for the source and target may be different, which produces an exogenous transformation. A transformation is defined at the meta-model level. Note that the meta-model of the languages involved in the transformation is referred to as the *transformation domain*. From a transformation definition, the transformation is automatically generated and is executed on any source model that conforms to the source meta-model. Both source and target meta-models, as well as the transformation specification, are themselves models, and conform to their respective meta-models: for meta-models, this is the classical notion of meta-meta-model; for transformations, a transformation language (or meta-model) allows a sound specification of transformations. Notice here that

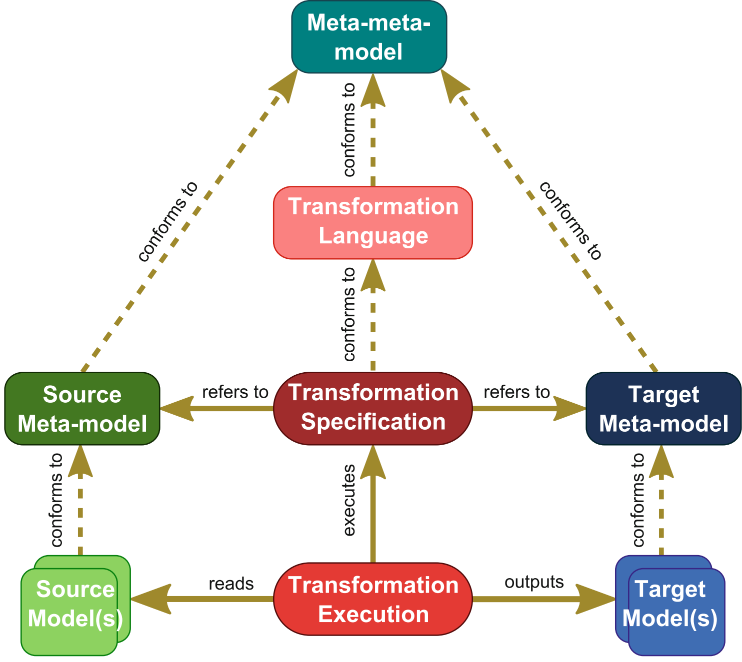


Fig. 1 Model transformation terminology

a transformation can also act on several source and/or target models. Figure 1 illustrates this definition.

Czarnecki et al. [6] list some of the features that an MTL can support. Lano et al. [24] compare some of the existing model transformation tools. A transformation is specified in the form of patterns which define the locations in the model where the transformation is applied. A pattern is the fundamental unit of a transformation. The very general area of model transformation spans many paradigms such as template-based, functional, imperative, and rule-based [6]. In this chapter, we concentrate our efforts on rule-based transformations, i.e., where the transformation units are rules. A rule is a declarative construct that dictates "what" shall be transformed and not "how." It consists of pre-condition and post-condition patterns. The pre-condition pattern determines the applicability of a rule: it is usually described with a left-hand side (LHS) and optional negative application conditions (NACs). The LHS defines the pattern that must be found in the input model to apply the rule. The NAC defines a pattern that shall not be present, inhibiting the application of the rule. The right-hand side (RHS) imposes the post-condition pattern to be found after the rule was applied. An advantage of using the rule-based transformation paradigm is that it allows specifying the transformation as a set of operational rewriting rules instead of using imperative programming languages.

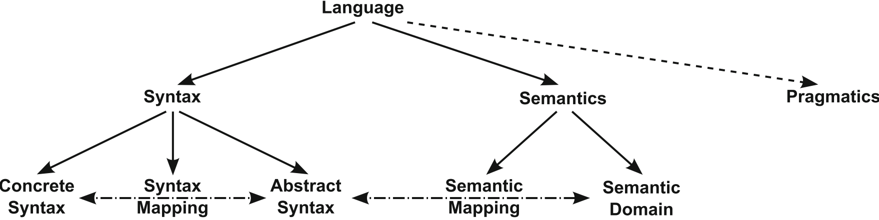


Fig. 2 Components of a modeling language

1.3 Modeling Language Engineering

As illustrated in Fig. 2, a software modeling language is defined by syntax and semantics [14]. The abstract syntax defines all the main concepts, elements, and their relationships. For example, these will be places, transitions, and arcs for a Petri net modeling language. The meta-model of a language consists of the abstract syntax and static semantics (e.g., places can hold a non-negative number of tokens in a Petri net). Therefore, the meta-model constrains the problem space to the essence of the domain it models. The concrete syntax defines the notations attached to each element of the abstract syntax. For example, a circle may define the graphical concrete syntax of a place in a Petri net. The syntax mapping thus plays the role of a renderer (concrete to abstract syntax) and parser (abstract to concrete syntax). The meaning of the modeling language is defined in a particular semantic domain. For example, the semantics of the Petri net language is defined using the reachability graph domain. The semantic mapping function assigns each element of the abstract syntax with elements in the semantic domain. The syntax of a language defines what constructs are allowed and its semantics define their meaning. Additionally, pragmatics of the language must be supplied in order to define the common uses and anti-patterns of the language. For example, a Petri net model consisting of only one place is valid, but useless.

1.4 Applying Modeling Principles to Model Transformation

This chapter lays the foundation for the engineering of MTLs, following a language engineering technique based on DSM concepts and MPM principles. There are several advantages for modeling explicitly all aspects of an MTL, as suggested in [19] for DSM. Another set of advantages can be found in [4], as summarized below:

- If we model transformations explicitly, we can re-use the same execution, generation and analysis techniques as for models. In this case, once the open

issues of maintenance, evolution, and verification tasks of models are solved, they can be applied on transformations for free.

- We can automatically synthesize code from transformation models to a target programming language and platform. The serialization of a transformation model can be used to load, save, and exchange transformation models seamlessly regardless of the tool used. The generated code can also be used as an interpreter of other models.
- If an appropriate meta-model is defined for an MTL, then one can automatically generate transformation-specific development environments.
- It becomes easier to explore the language design space by making alterations to the control flow, mapping, and pattern specification parts of the language. Obviously, this requires modeling the respective semantics, but once available, alterations to the syntax and semantic definitions of such transformation (meta-) models should be easier to perform than the respective changes in a code base.
- When transformations are modeled explicitly, then transformations can be transformed cleanly through higher-order transformation. This can automate the optimization, analysis, and integration of MTLs.

2 Modeling the Syntax

The first step is to design the syntax of the MTL, which reveals the concepts, components, and their relationships in an MTL, as well as visual notations adapted to the domain on which the transformation shall be applied. We model the syntax of an MTL with an *explicit* meta-model that is a best fit for domain-specific transformations.

2.1 Models, Meta-Models and Transformations

The diagram in Fig. 3a depicts the relations between a transformation and the artifacts associated with the transformation. T is the *operation* that transforms a model M into a model M'. Each model conforms to its respective meta-model MM_1 and MM_2. M_T models this transformation and, conversely, T executes M_T. In fact, M_T is a *model* of a transformation that transforms any model of MM_1 into a model of MM_2. MM_T is a *meta-model* of all transformations that transform any meta-model. Since everything is modeled explicitly, *MMM* is the *meta-meta-model*, i.e., it is the meta-model of the language used to describe meta-models. Typically, *MMM* conforms to itself in a sound bootstrapped environment. This explicit point of view on models of transformations is compatible with the model-driven architecture (MDA) meta-layers [21] depicted in Fig. 3b. It places a transformation at the level of real systems (the M_0 layer), a model transformation (or transformation model) at the instance level (the M_1 layer) and the MTL at the level of UML class diagrams

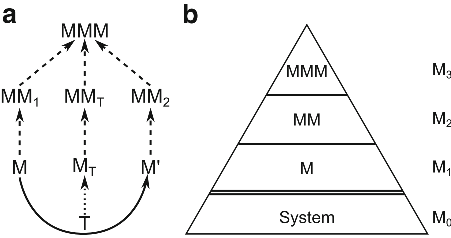


Fig. 3 **(a)** Meta-modeling of model transformations and **(b)** the MDA meta-layers

used to define a meta-model (the M_2 layer). In MDA, the M_3 layer would represent the meta-model of UML, i.e., the meta-object facility (MOF) [12].

The contribution of what follows is an approach that provides transformation development environments that are customized to the specific domain of application. Therefore, the focus is on MM_T, the meta-model of transformations, to provide a general solution. The meta-model MM_T of a model transformation can be partitioned into three sub-models:

- The meta-model of the *pattern language* (MM_{PL}) defines the language of the patterns or model fragments specified in the pre- and post-conditions of transformation rules. It highly depends on the input and output languages of a transformation.
- The meta-model of the *transformation units* (MM_{TU}) defines the language of the individual building blocks of an MTL (e.g., rules, helper functions, relations, and modules).
- The meta-model of the *scheduling language* (MM_{SC}) defines the language of the execution logic of a transformation (e.g., a programming language, a workflow language, or a modeling language).

2.2 Pattern Language

Unlike the mapping and control aspects of an MTL, its pattern specification sub-language depends on other languages that represent the domain of the transformation. The input and output languages of a transformation determine which pattern specifications for the pre-condition and the post-condition can be considered well-formed. The underlying assumption is that the pattern specification language should not be generic to fit all possible input and output languages, but specifically tailored to the input and output languages, involved.

A generic pattern specification language is the most economical solution because it can be re-used to specify the patterns of a transformation applied on any domain. However, this makes the concrete syntax of the patterns inadequate for the domain. Most tools use a UML object diagram-inspired concrete syntax for generic pattern languages as illustrated in Fig. 4a, which requires additional expertise and

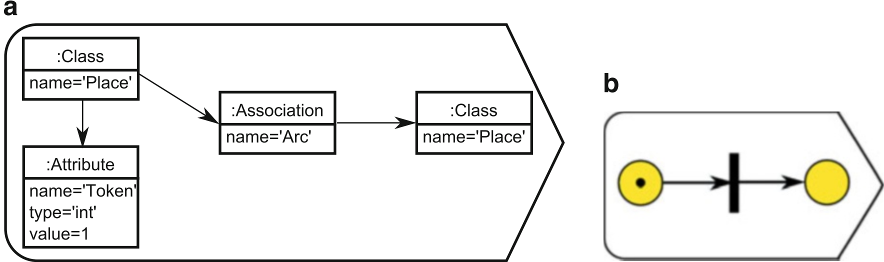


Fig. 4 In **(a)** a valid generic MOF-based pre-condition pattern of an invalid Petri net. In **(b)** a valid domain-specific pre-condition pattern for a valid Petri net

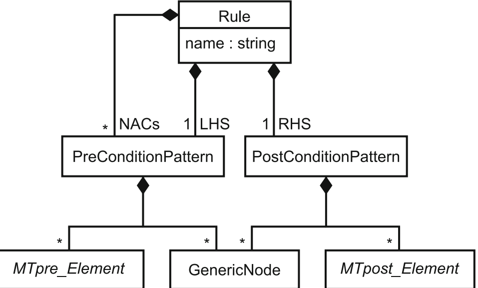


Fig. 5 The meta-model of a rule

knowledge for the domain expert modeler. Furthermore, it allows the modeler to specify patterns that may never occur. For example, two Petri net places are connected with an arc in Fig. 4a. Although relating two classes with an association is valid in UML, it is invalid in a Petri net, since places shall only be connected to transitions and vice versa. In contrast, one can consider a pattern specification language that is customized to the input/output languages involved, as in Fig. 4b. In that case, it excludes patterns that do not have a chance of being matched from those that can be expressed. It also provides a concrete syntax adapted to the source/target languages, which is especially relevant for domain-specific languages. However, this approach imposes more work for the tool builder because it requires the definition of a new pattern language for each transformation involving different domains.

The original language definitions (meta-models) cannot be used for defining the well-formedness of pattern specifications. Instead, transformation pattern specifications represent fragments of models from that language. A distinct meta-model for transformation rules and the patterns specified in them is required. Figure 5 introduces a meta-model of a rule-based transformation unit, which refers to pre- and post-condition patterns, as well as the pattern elements they contain. When adapting transformation languages to specific input and output languages, one needs to tailor these pre- and post-condition patterns so that they are fit to be used for the respective input and output languages. We obtain the required tailored

pattern specification meta-models by starting with the original language meta-models and then subjecting them to a number of changes. In previous work [23], we have shown how to semi-automatically generate a domain-specific pattern language from the input/output meta-models. This meta-model metamorphosis, called the *RAMification process*, involves three steps: Relaxation, Augmentation, and Modification. The process is specific to languages for which the meta-model is defined by a UML class diagram.

Relaxation. This step relaxes the constraints imposed by the meta-model of the domain. For example, it allows the instantiation of classes that were originally abstract. This allows a single rule to match model elements of any of its sub-types. It reduces the minimal multiplicity of every association end (e.g., a 1..2 multiplicity is relaxed to 0..2). This allows the presence of isolated association elements in a pattern. Any additional constraint (possibly defined in OCL) needs to be removed or preserved, depending on the static semantics of the pattern language. This decision is not automated and requires manual filtering.

Augmentation. This step augments the resulting meta-model with additional information. All the meta-model classes and associations are integrated in the rule meta-model of Fig. 5 through inheritance. This results in two meta-models: one for the pre-condition patterns and one for the post-condition patterns. Transformation-specific properties and constraints are also added to the meta-models such as labels of pattern elements, and parameter passing between different rules.

Modification. This step performs some further modifications on the resulting meta-model. Namespaces are updated accordingly. The types of the attributes are modified as well. For pre-condition classes, all attributes are of the type of the constraint language (e.g., OCL). For post-condition classes, all attributes are of the type of the action language (that is not bound to a particular language, e.g., imperative OCL or Kermeta [30]). The concrete syntax of the original meta-model is preserved as much as possible. However, associations rendered as topological constraints (overlap, positioning) and invisible associations are replaced by visual arrows. This can later be altered by the modeler as desired.

The RAMification process generates two meta-models, one for the pre-condition pattern language and the other for the post-condition pattern language. Each of these pattern meta-models is based on all the meta-models of the languages involved in the transformation.

We have integrated the RAMification process in the meta-modeling process of our tool AToM^3 [25]. After defining the meta-model of the source and target domains of the transformation, the transformation developer can generate the RAMified meta-models, one for the meta-model of the pre-condition patterns and one for the meta-model of the post-condition patterns. The meta-model of the transformation units can be loaded, consisting of LHS, RHS, and optionally NAC components of a rule or query. Patterns can then be specified in each component as illustrated in Fig. 6. This process illustrates how a modeling environment is

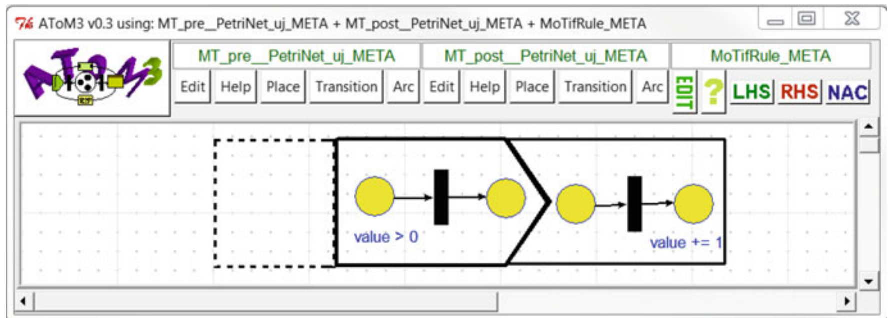


Fig. 6 A transformation rule model in AToM^3

automatically synthesized for the design of the transformation rules that are specific to the problem domain.

2.3 Transformation Units

The typical transformation units of an MTL are rules, queries, their compositions through modules/packages, and helper functions that are re-usable in different rules or transformations. A query is similar to a rule in the sense that it binds part of the input model to its pre-condition pattern. However, queries are side-effect free. That is, the model remains identical after a query was executed. The output of a query is an out-place view of the input model, which can be a snapshot of a sub-set of the input model or an abstraction of some of its properties through aggregation. Examples of queries can be found in the QVT specification [13] or attribute helpers in ATL [17]. Helper functions are typically rules or queries with the purpose of encapsulating parts of a transformation unit that appear in multiple locations of a transformation. Examples of helper functions are operation helpers in ATL.

In previous work [37], we identified the primitive building blocks of transformation units. T-Core encapsulates this minimal collection of transformation primitives. In [35], we showed how to re-construct the transformation units of a dozen existing MTLs with T-Core. The primitives offer the following features:

- *Pre- and post-condition patterns* allow one to specify a rule declaratively in a relational (QVT-Relation) or operational (graph transformation) paradigm.
- *Matcher* binds model elements that satisfy a pre-condition pattern.
- *Rewriter* transforms the host model to satisfy the post-condition of a rule.
- *Resolver* validates whether a rule has been consistently applied in order to detect inter-rule conflicts and resolves them.
- *Manipulation of matches* to iterate through them and roll back to previous match states.
- *Control of the flow* of rule applications by offering choices and concurrency.

- *Composition* mechanisms to provide structure, re-use, and encapsulation of the above primitives.

The data exchanged between T-Core operators are *packets*, which carry all the necessary information required by every operator. T-Core also supports exception handling. Exceptions are modeled explicitly as messages that the operators can exchange. Each operator is customizable through pre-defined parameters. All primitives conform to a common API making their interleaving independent from each other. In addition, the composition mechanism allows transformation developers to build custom transformation units. For example, a simple transformation rule consists of a matcher that first computes a binding of the pre-condition pattern over the input model. The iterator selects one of the possible matches. Finally, the rewriter modifies the model (with the usual CRUD operations [20]) so that the matched sub-models satisfy the post-condition pattern.

T-Core empowers the transformation developer to build transformation units that are specific to the problem the transformation will solve. General-purpose transformation languages, such as QVT or ATL, may encumber the developer with unneeded features. This adds complexity to the design of the transformation that may lead to design errors and even reduce productivity [19]. In contrast, MTLs based on T-Core, using RAMification, enable the design of transformation models that are tailored to the problem domain.

2.4 Scheduling Language Meta-Model

The third part of the meta-model of an MTL is the scheduling language. It can be a programming language (e.g., Java [9]) or a modeling language (e.g., UML Activity diagrams [28], or Colored Petri nets [40]). We now present MoTif, a DSL for the scheduling of model transformation rules.

MoTif is a modeling language for designing model transformations based on graph transformation [8]. This language is engineered following MPM principles where everything is explicitly modeled at the most appropriate level of abstraction using the most appropriate formalism. Therefore, its meta-model, depicted in Fig. 7, consists of three parts: the pattern language, the scheduling language, and the transformation units.

The pattern language is adapted automatically to the domain of application of each transformation following the RAMification process described earlier. It is integrated in the meta-model of MoTif by extending the Pre- and PostCondition-Pattern classes through inheritance, as shown in Fig. 5. MoTif is a controlled, timed graph transformation language. It offers a clean separation of the transformation units from the structure and flow of execution of the transformation. Rules and queries are the supported transformation units. Transformation units are embedded in AtomicRuleBlocks that are part of the scheduling language. This language mainly consists of RuleBlocks. A RuleBlock can be either atomic or composite

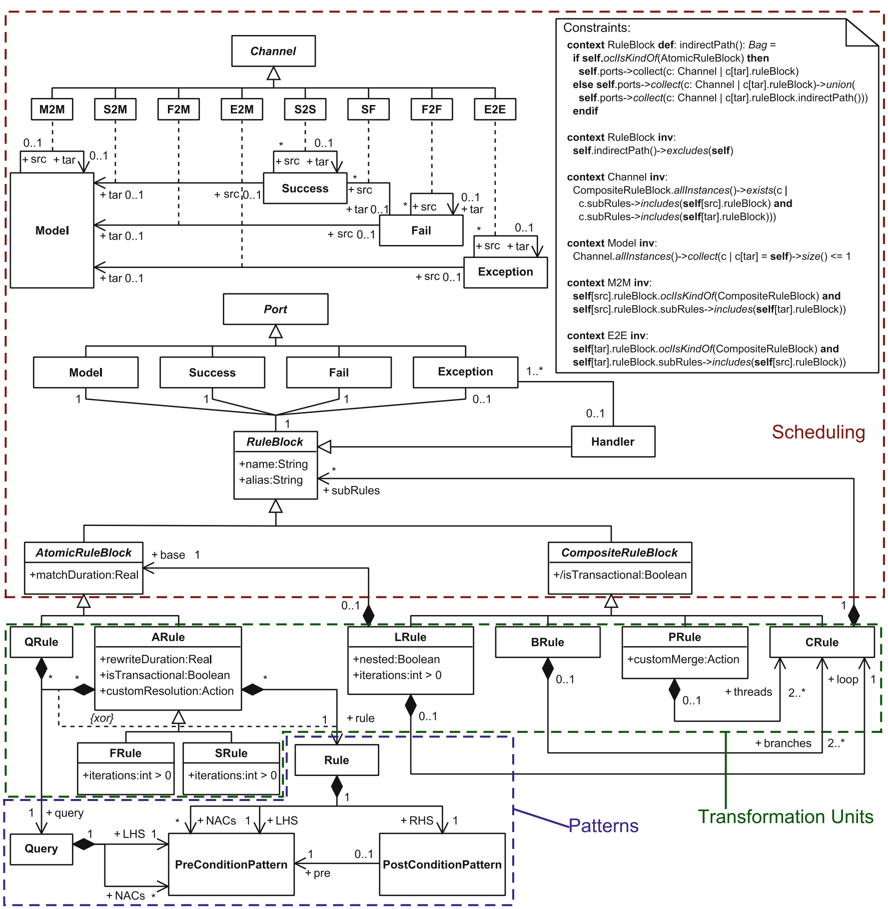


Fig. 7 The meta-model of MoTif

(CompositeRuleBlock). In the atomic rule blocks, an ARule ("Atomic Rule") encodes a rule and a QRule ("Query Rule") encodes a query. Composite rule blocks are used to modularly encapsulate other RuleBlocks (atomic or composite). Some composite blocks express advanced control flow structures, such as branching, looping, and parallelism. RuleBlocks have ports that can be connected via channels. Model is the only input port and Success, Fail, and Exception represent output ports. Port channeling induces an ordered application *sequence* of rule blocks.

In the MoTif visual modeling language, the concrete syntax of an ARule is a single rectangle frame as depicted in Fig. 8. The top triangle on a rule block is the Model input port. The bottom-left tick symbol is the Success output port and the bottom-right "X" symbol is the Fail output port. Conceptually, the input model is received on the Model port and, if the application of the rule is successful, the resulting model is output through the Success port. However, if the pre-condition

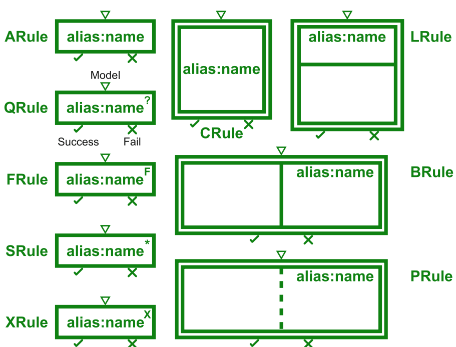


Fig. 8 The different rule blocks in MoTif

pattern is not satisfied, the original model is output from the Fail port. The QRule has a similar graphical syntax with a question mark symbol on the top-right of the rectangle. The transformation engineer can specify a time duration for the matching phase for both atomic rule blocks. In the case of an ARule, the duration for the rewriting phase can also be specified.

A MoTif sub-model encoding transformation units can be part of a CRule ("Composite Rule"). CRules are visually depicted by a double rectangle frame. The same ports appear on both atomic and composite rule blocks which implies that they can be used interchangeably to build complex *hierarchical* transformation models modularly.

Iterative rule application is possible with variants of an ARule. The FRule ("For all Rule") applies the transformation rule on *all* matches of the pre-condition pattern (in an arbitrary, but deterministic and repeatable order). The maximum number of iterations is parameterizable. The matches are assumed to be parallel independent [8]. Two matches are parallel independent if no overlapping matched element is modified (node deletion or attribute modification) by the rule when applied. If they are not, the transformation designer can specify a resolution function to resolve the conflicts. The FRule is represented using the same concrete visual syntax as an ARule, annotated with an "F" in the top-right corner. If the maximum number of iterations is not infinite, the positive integer appears in the top-left corner.

Another variant of the ARule is the SRule ("Star Rule"). It is applied sequentially as long as the pre-condition pattern is satisfied in the model. That is, after the model received is matched and transformed, the resulting model is then matched again by the same rule. This continues until no more matches can be found in the resulting model. Care should be taken when using this construct as it may result in an infinite loop. When combined with pivot[1] passing, the SRule applies itself *recursively*. The SRule is represented using the same concrete visual syntax as an

[1] A pivot acts like a parameter for transformation rules. It allows certain elements bound in one rule to be passed to another rule.

ARule, annotated with an asterisk in the top-right corner. If the maximum number of iterations is not infinite, the positive integer appears in the top-left corner.

While iteration involves a single rule block, *looping* allows one to iterate over multiple rule blocks. This is possible with the LRule ("Loop Rule"). It consists of an atomic rule block as base and a CRule as loop body. The LRule applies the rules of the loop body iteratively for every match found in the base rule (c.f. Fig. 7). The LRule has different variants depending on the type of the base rule block and whether pivots are used in the patterns, such as rule nesting and indirect recursion. For example, if the base is an FRule, then all matches of the base are first found. Then, at each iteration, the rewriting part of the base is applied. In that case, the rule block is referred to as an LFRule. If the base is an SRule, the behavior is similar except that the matches are re-evaluated at the end of each iteration. In that case, the rule block is referred to as an LSRule. The concrete syntax of an LRule is the same as a CRule, but a horizontal solid line separates the base compartment from the loop compartment.

In graph transformation, it is sometimes desirable to have many rules match, but let only one be applied. MoTif introduces the BRule ("Branch Rule") block which allows *branching* of rules. Its purpose is to receive a model, through its Model port, and send it to each branch. However, only one branch—of those that successfully found a match—is selected to continue executing the transformation. Visually, a BRule is similar to a CRule, but the rectangle is partitioned by vertical filled lines to separate the branches, each branch being a CRule in its own right.

MoTif allows rules to be applied in *parallel* with the PRule ("Parallel Rule"). This leads to what we call "threads" of rule applications. Each thread is applied concurrently, independently from each other. The output of a PRule is a single model "merged" from the result of each thread. The threads are assumed to be sequential independent [8]. The transformation designer can specify a merge function to merge the graphs in the case of conflict. The PRule's parallel execution requires special care. Visually, a PRule is similar to a BRule, but vertical lines that separate the threads are dashed. Each thread is also a CRule.

When the `isTransactional` flag of a rule block is activated, its behavior is extended with memory capacity, which provides *back-tracking*. We denote such a rule block by XRule ("Transactional Rule"). For composite rule blocks, `isTransactional` is set to true if and only if the first sub-rule is transactional. Through transactional rules, MoTif also allows to model *recursion*, as explained in [38]. A transactional rule block has the same concrete visual syntax as the rule block, with an "X" appended in the top-right corner.

Note how the meta-model of MoTif does not include any information about the models processed. This is because MoTif is a language that constrains the transformation modeler to only focus on describing the scheduling part of model transformation.

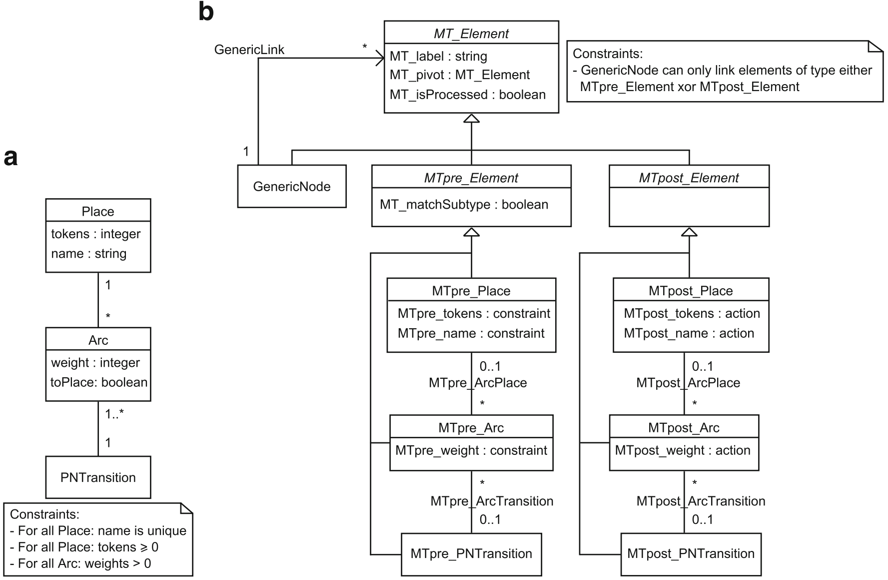


Fig. 9 **(a)** The original meta-model of Petri net rules and **(b)** its RAMified version

2.5 Example: Petri Net Simulator

We illustrate the use of MoTif by modeling a Petri net simulator. Figure 9a shows the original meta-model of the language and Fig. 9b shows the pattern meta-model derived from it following the RAMification process. The relaxation step reduced the multiplicities of the associations between arcs, places, and transitions to be minimally 0. It also removed the three original constraints. The augmentation step added transformation-specific attributes such as labels. It also split the meta-model into pre- and post-condition meta-models. The modification step renamed each class to reflect whether it is to be considered as a pre- or a post-condition element. Attributes are re-typed as constraints and actions, respectively.

We simulate the execution of a Petri net execution by using a small set of transformation rules (see Fig. 10a), transforming Petri nets to Petri nets. We are able to express the operational semantics of Petri nets in just four simple rules because of the expressiveness of MoTif control structures. The control structure is shown in Fig. 10b. The control structure makes it particularly easy to find an enabled Petri net transition, i.e., one which can fire. Such a transition needs sufficiently many tokens at *each* of its incoming transitions. One naive solution for finding enabled transitions is to just specify all possible patterns to be found. Alternatively, this can be solved provided that the pattern specification language uses intentional specifications to allow referring to sub-graphs of arbitrary size. However, the most elegant solution is to iterate through all transitions until one has been found that does *not* satisfy the pattern of a *non*-firing transition.

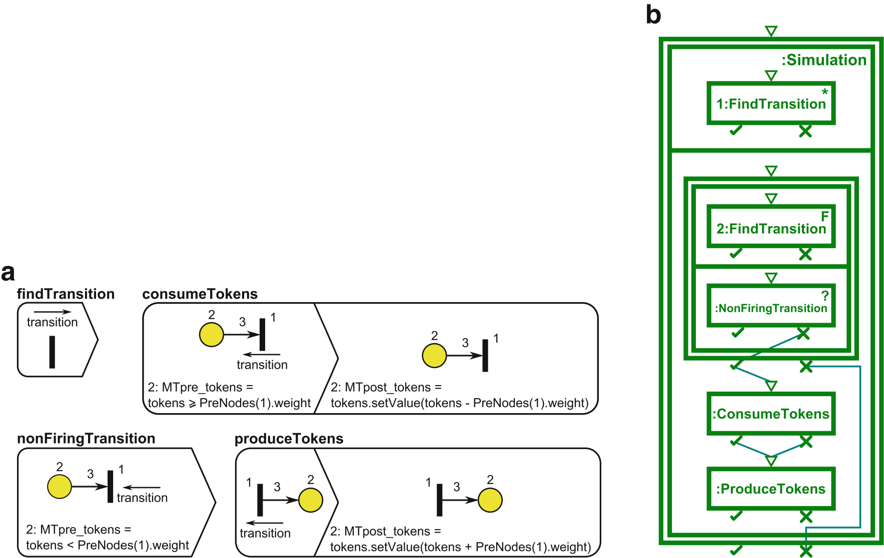


Fig. 10 The operational semantics for Petri nets: the rules in **(a)** and the control flow in **(b)**

The behavior of the transformation model is as follows. The outer-most rule block is an LSRule called Simulation—since the base rule block of this LRule is an SRule. Although the base is a query, it is nevertheless encapsulated in an SRule (with no rewriting phase) to recursively execute the transformation. First, 1:FindTransition looks for one transition. The transition found is assigned to a pivot called `transition`. In the loop part of the LSRule, an LFRule ensures that only firing transitions will be processed. This is done by verifying if a given transition (assigned to the pivot) cannot fire. If the NonFiringTransition rule is not applicable on that transition, then it is firing. This interruption in the loop is represented by the channel from the fail port of the NonFiringTransition QRule to the success port of the enclosing LFRule. When a firing transition is found, it is assigned to a pivot called `transition`, replacing the former transition, to be processed by the subsequent rules. Then, tokens are transferred along this transition as depicted by rules ConsumeTokens and ProduceTokens. After that, the base rule block of Simulation is applied again recursively, by re-matching the new model looking for a transition. This control flow continues until no more transitions are fire-able.

3 Modeling the Semantics

In the previous section, we showed how to model the complete syntax of an MTL and then demonstrated the execution of a model transformation in that MTL. We now describe precisely how to model the MTL's semantics.

3.1 Semantic Domain and Mapping

3.1.1 Semantic Domain

The semantic domain of an MTL is a language that is used to define the meaning of its transformation models. The semantic domain is often confused with its implementation, in which case the semantic domain is a programming language (as is the case for ATL). Transformations are operations performed on models. It is therefore natural to opt for a language that can define operational semantics conveniently. It should nevertheless be a formal, precise, and unambiguous language. Popular candidates are programming languages, structured operational semantics [3], or state-automata languages (e.g., Petri nets [40], abstract state machines [2]). We claim that the semantics of an MTL should not be defined in terms of a programming language, because this is implementation-specific, which is counter to the goals of MDE.

In the case of MoTif, the semantic domain of its scheduling part is the Discrete Event System Specification (DEVS) formalism [41] together with T-Core. A DEVS model consists of atomic processes that are similar to timed automata. The behavior of atomic processes is independent from each other. They exchange events through channels between their ports and may generate new events or consume events. The parallel composition of atomic processes is defined by coupled models, which enables re-use of sub-models in a modular way.

DEVS is an attractive semantic domain for graph transformation languages as explained in [36]. Since DEVS inherently allows one to build hierarchical models, the transformation language becomes highly modular by re-using specific components of a transformation. Another side effect of using DEVS is the explicit introduction of the notion of time in model transformations. This allows one to model a time-advance for every rule, as well as to interrupt (preempt) rule execution. Another advantage is that the behavior of DEVS models is defined by a platform-independent simulator for which several efficient implementations exist. Also, the same DEVS model can be executed on a sequential, parallel, or distributed environment.

3.1.2 Semantic Mapping

Mapping the meta-model of a modeling language to a particular semantic domain is the most complex task in the design of the MTL. It is the core of the language as it defines the meaning of every element that the modeler uses to build transformations. The complexity of the semantic mapping function lies in the typical domain mismatch between the syntactic domain and the semantic domain. Often—at least in DSLs—the syntax is "high-level" and is as close to the domain abstraction as possible. In contrast, the semantic domain is often a formal language with no straightforward resemblance to the syntactic domain. Our solution is to divide the

semantic mapping into several steps. At each step, a semantic concept is mapped to an intermediate language until a mapping to the semantic domain can be "easily defined." We *model* these "smaller" semantic mapping functions by higher-order transformations (HOTs) [39]: transformations transforming transformation models. Note that this is only possible if the language of the semantic domain is also modeled with an explicit meta-model.

Consider the MoTif language. From a syntactical point of view, it is a domain-specific language for modeling transformations with T-Core primitives. However, its semantics rely entirely on DEVS. Thus, the transformation engineer is required to have expertise in the DEVS formalism, which is rarely the case. Designing transformation models with DEVS is not a trivial task. To leverage this complexity, we introduce MoTif-Core as an intermediate language to facilitate this mapping. MoTif-Core is intended to offer a common platform for model transformation languages that are at a more appropriate level of abstraction. MoTif is therefore a transformation-specific language whose syntax abstracts away T-Core constructs and whose semantics is defined in terms of MoTif-Core.

3.2 MoTif-Core as an Intermediate Semantic Domain

As an intermediate language, MoTif-Core is an extension of the DEVS formalism combining it with T-Core primitives. MoTif-Core is also an MTL with a syntax and a semantics. Its meta-model is described by the UML class diagram in Fig. 11. The semantic domain of MoTif-Core is the DEVS formalism and the semantic mapping function is described using set theory and mathematical functions. We refer readers interested in the semantic mapping to DEVS to Chap. 7 of [35]. The following focuses on the syntax of MoTif-Core and the semantics is explained informally.

The T-Core primitives can be found encapsulated in the state of different atomic DEVS AtomicPrimitive elements. Those classes are prefixed by "TC," depicting that they are semantically identical to their T-Core counterparts (e.g., in MoTif-Core, TCMatcher represents the Matcher from T-Core). They exchange packets, modeled as DEVS events, which encapsulate the model to be transformed. When a RulePrimitive element receives a packet from an inport, its external transition function is triggered and invokes the appropriate method of its corresponding T-Core primitive according to the activated inport. After the packet (or any other event) is processed by the T-Core primitive, it is sent via an outport of the RulePrimitive element: the output function ensures the conversion from a T-Core message to a MoTif-Core event. As both functions modify the state of the MoTif-Core element, the time advance function defines the delay for which it shall remain in the new state. Each RulePrimitive element has a specific set of in/outports. For example, the Matcher has APacketIn and ACancelIn inports from which packets or cancel events are received, respectively. It also has a ASuccessOut and AFailOut outports from which packets are output depending on the semantics of whether the encapsulated T-Core primitive "was successfully

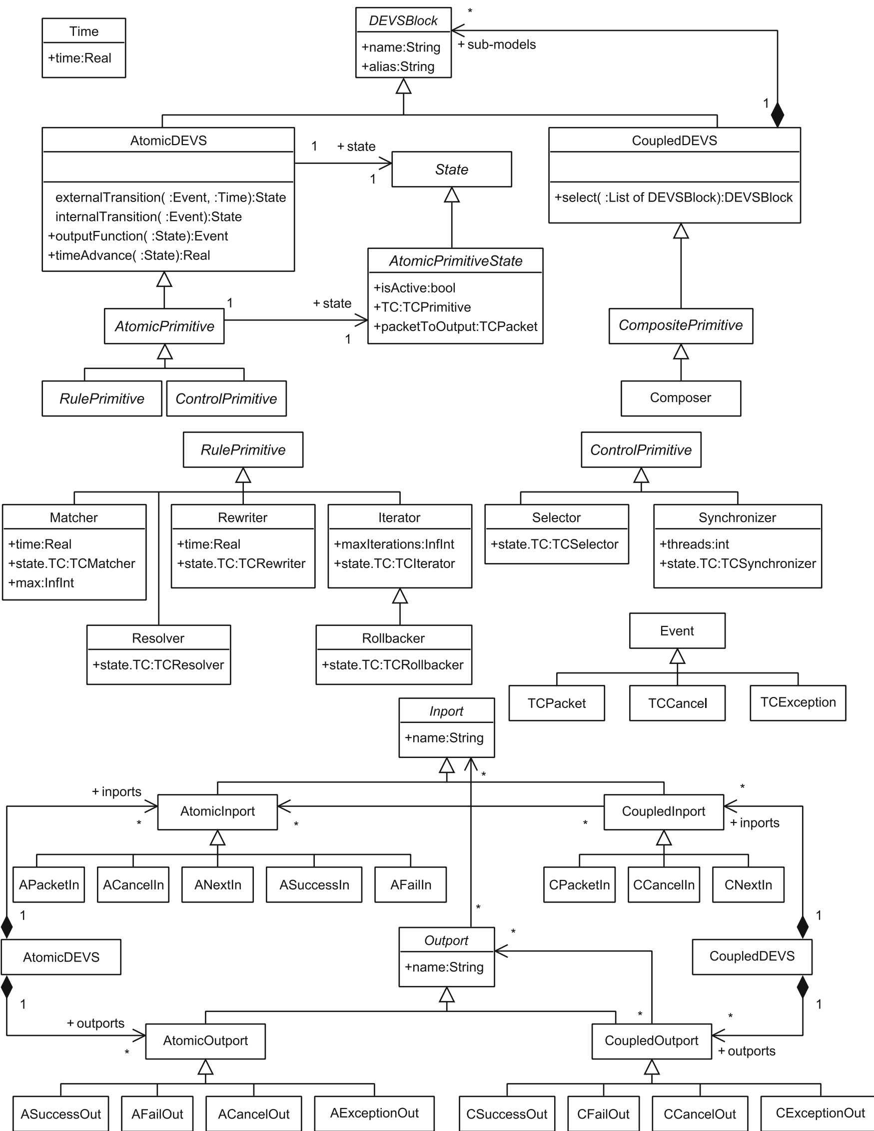


Fig. 11 The meta-model of MoTif-Core

applied" on the packet. For instance, if a match of the pre-condition was found in the packet, the T-Core matcher was successful and the resulting packet is output from ASuccessOut. In MoTif-Core, the Composer is coupled to the DEVS model and hence inherently composes other elements. It specifies the connection between the different in/outports to ensure a proper flow of the transformation.

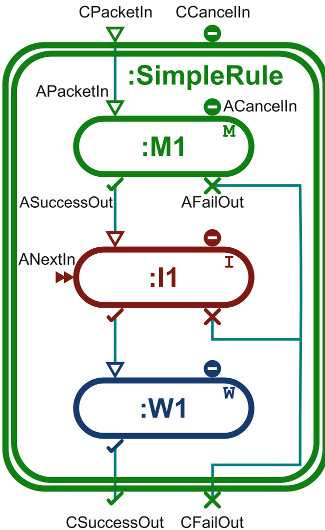


Fig. 12 A Composer representing a simple transformation rule

To illustrate the concrete syntax and behavior of MoTif-Core, Fig. 12 describes how a simple graph transformation rule is modeled. In graph transformation, a rule behaves as follows. Given a transformation rule pattern, the rule first looks for an occurrence of its LHS pattern to match in the input graph. If a match is found, the rule transforms the graph by rewriting the matched sub-graph, resulting in the RHS pattern. The graphical syntax of MoTif-Core primitives is a rounded rectangle labeled on the top-right by its type (M for Matcher, I for Iterator, W for Rewriter). Inside it, the optional `alias` is followed by a colon and then a `name` identifying the T-Core primitive. A Composer is represented by a double-lined rounded rectangle. A line depicts a channel connecting ports. The Composer has three inner models: a Matcher, an Iterator, and a Rewriter. Its behavior is as follows. The packet the SimpleRule receives via its CPacketIn inport is first sent to the Matcher. When the Matcher receives the packet, any occurrence in the graph of its pre-condition pattern is stored in the packet. After a certain delay specified by its time advance, if a match is found, the Iterator receives the modified packet output from the Matcher and selects one match (in this case the only one). Then, the Rewriter receives the packet output from the Iterator and transforms the graph according to its post-condition pattern applied on the selected match (specified in the packet). After a certain delay, the resulting packet is sent to an outport of the Composer. In the case of a successful application, the newly modified packet is sent through the success outport CSuccessOut. If the Matcher was unable to find any matches, or if the Iterator has exceeded the number of iterations (to select a match), the packet is sent through the fail port CFailOut, depicting that the SimpleRule was not applied.

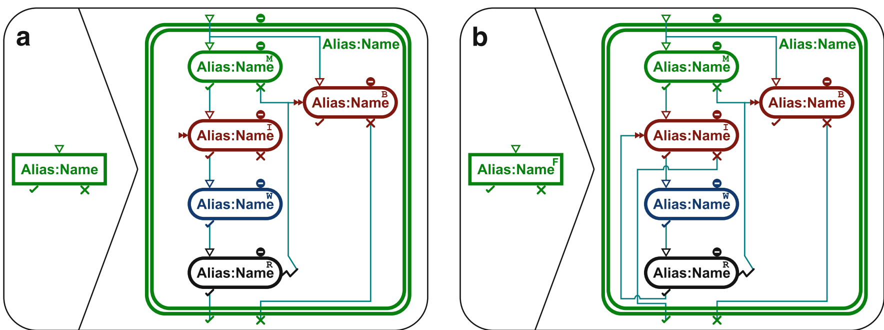


Fig. 13 The ARule **(a)** and FRule **(b)** in MoTif and their equivalent MoTif-Core models

3.3 *Semantic Mapping Function as a Higher-Order Transformation*

We have described the abstract and concrete syntax of MoTif. We have also described its semantics domain, MoTif-Core, with its concrete syntax and whose semantics is defined by a mapping onto DEVS. What remains is the semantic mapping of MoTif. It must be injective since every MoTif construct shall correspond to a unique MoTif-Core model. We propose to describe this mapping as a model transformation specified in MoTif. It is therefore a HOT transforming one transformation language (MoTif) into another (MoTif-Core). The meta-model of the patterns of this transformation consists of the RAMified meta-models of MoTif and MoTif-Core. In this subsection, we outline the main transformation steps of the semantic mapping, focusing on the ARule and the FRule. The complete transformation can be found in [35].

Every RuleBlock in MoTif is mapped onto a Composer in MoTif-Core. The *alias* and *name* parameters of the Composer are the same as those of its RuleBlock counterpart. The Model port is mapped to the CPacketIn port of the Composer, the Success port to the CSuccessOut port and the Fail port to the CFailOut port.

The ARule is the simplest transformation unit with side effect. When an ARule receives a model input from the Model port, it searches for one occurrence of its LHS in the input model. If a match is found, it is transformed according to the RHS of the rule. Figure 13a illustrates how an ARule is mapped onto a MoTif-Core Composer. The rule in the figure describes the connection topology of the Composer's sub-models. An ARule behaves similarly to the simple rule described above. However, a Resolver (rounded rectangle annotated with an R) is added in case a pending match in the packet conflicts with the current rule application. Visually, the zigzag on its right depicts the AExceptionOut port from which an exception event encapsulating the packet is output if the Resolver cannot resolve the conflicts. To ensure the atomicity of the graph transformation rule, a Rollbacker

(rounded rectangle annotated with an B) is added to the MoTif-Core model. It ensures that if the rule is not applied, the packet will be restored to the state it was in before entering the Composer.

Recall that the FRule is an ARule that applies its transformation phase on all the matches found before the new model is output. As shown in Fig. 13b, the matching phase is performed only once and, after the match is rewritten and validated, the packet is sent back to the Iterator that will select another match to process. Note that the Iterator failing (i.e., outputs a packet from AFailOut) means that the Matcher has successfully found all matches in the host graph and there are no more matches left to process. In this case, the FRule will successfully output the new packet. If, however, the Rewriter or the Resolver fails during one of the iterations, all the modifications that were performed in this Composer are discarded through the Rollbacker. The user can control the number of times the rule encoded in the FRule is applied. If $iterations = \infty$, it will be applied on all possible matches. The order in which matches are processed is non-deterministic as it relies on the behavior of T-Core's TCIterator. Also, the TCResolver will by default fail if any two matches overlap. This will result in discarding all previous transformations performed by this FRule. This appears at first to be an excessive overhead because the confluence of the matches could have been detected prior to the execution, e.g., through the computation of critical pairs [16]. However, the latter approach, in addition to not scaling well, may sometimes be too conservative leading to false positives (an example is given in [15]). This is overcome in MoTif by letting the user override the validation criteria *customResolution* of the FRule.

4 Deployment

MoTif is a completely modeled MTL. Its syntax is convenient to use for a transformation engineer in the sense that it only contains artifacts specific to transformations (unlike MoTif-Core). Its semantics is also modeled explicitly since it is mapped onto the MoTif-Core modeling language and this mapping is modeled as a transformation. Figure 14 illustrates the different language layers MoTif relies on. MoTif is a "syntactic sugar" language of MoTif-Core, which consists of the core elements of the language. The former simply defines a more user-friendly syntax encapsulating the different transformation operators provided in the latter language. MoTif-Core combines T-Core and DEVS, both running on a model-aware virtual machine. They are expressed in a neutral target language as defined by the AToM^3 Redux Kernel (ARK) [7], which represents the meta-meta-modeling layer in AToM^3. The tool is implemented in Python. The DEVS virtual machine allows executing MoTif transformations.

Figure 15 shows the framework in which MoTif transformation models are executed. MoTif is a formalism defined in AToM^3 as a domain-specific language. To define a transformation, the transformation engineer generates a modeling environment for designing rules by applying the RAMification process on the source

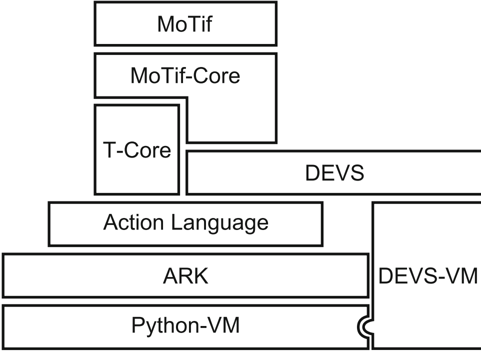


Fig. 14 The architecture of the MoTif language

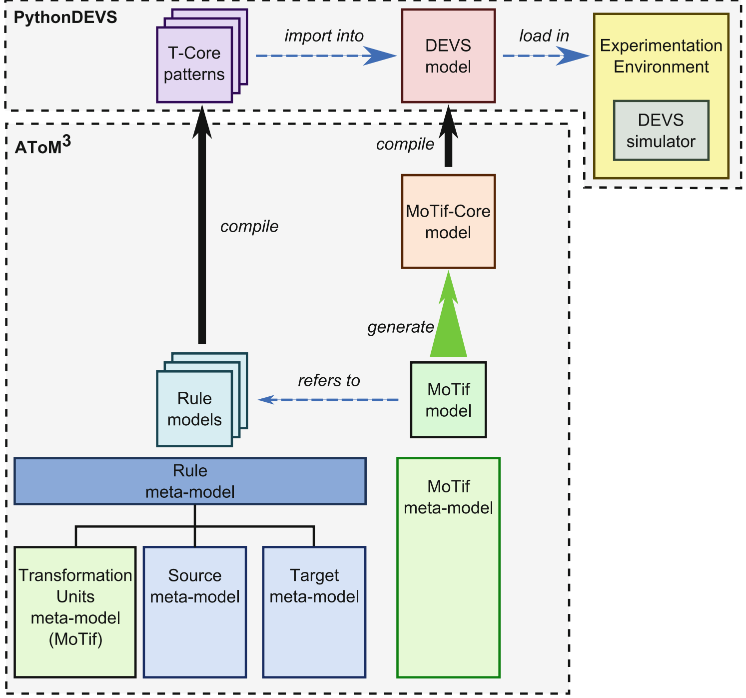


Fig. 15 The MoTif execution framework

and target languages. The result is automatically combined with the transformation unit part of the meta-model of MoTif and produces a customized meta-model for the patterns of the transformation. On the one hand, the transformation engineer defines rules, queries, and their patterns in the modeling environment. They are then automatically compiled into T-Core patterns. On the other hand, the transformation engineer specifies the control flow of the transformation by designing a MoTif model. Then, the MoTif model is transformed into an equivalent MoTif-Core model.

The resulting model is further compiled into a PythonDEVS [5] model (a Python implementation of DEVS). The generated T-Core patterns are integrated in the DEVS model through package imports. A PythonDEVS simulation environment is also generated from the MoTif-Core model. It allows one to interact with, execute, and debug the PythonDEVS model.

5 Related Work

The first mention and motivation to treat transformations as models and not as programs appeared in [4]. The idea of defining the semantics of a model transformation language at a higher level of abstraction (such as MoTif) with respect to another model transformation language at a lower level of abstraction (such as MoTif-Core) has recently gained popularity. For example, the semantics of QVT-Relations is defined in terms of QVT-Core [13]. This semantic mapping is defined by a QVT-Relations transformation. In the case of MoTif, the mapping is defined as a higher-order transformation which is, in turn, expressed in MoTif. Since MoTif is formally defined in terms of DEVS, T-Core, and graph transformation, the semantic mapping of MoTif to MoTif-Core is formally defined.[2]

MoTif was introduced in [38] as a new general-purpose transformation language whose rule scheduling semantics is based on DEVS. The semantic mapping from MoTif to DEVS was defined using set theory and morphisms, which the transformation developer may not have familiarity. In our current approach, we explicitly *model* the mapping to DEVS instead. We believe this provides a more accurate and simpler documentation. It also facilitates the evolution and maintenance of the language. For example, when a new version of MoTif is deployed, all existing transformation models conforming to the previous version can be migrated automatically by a HOT. The proposed definition of the semantics allowed us to detect errors in the original mapping to DEVS, which were hard to detect in their initial form.

Several approaches define transformations using concrete syntax. For example, Baar and Whittle [1] show how the concrete syntax of visual modeling languages is adapted for the specification of transformation rules. Rumpe and Weisemöller [32] achieved a similar goal for transformations in textual syntax. However, in both cases, the adaptation of the syntax is performed manually as opposed to the approach presented in this chapter.

There have been previous efforts that have provided a means to describe a model transformation using terms that are tied to domain concepts from a specific meta-model, rather than abstract modeling concepts from the meta-meta-model. One of the earliest examples of this type of work was on ANEMIC [31], which was a specific approach for writing model interpreters for GME [27]. The typical way to write a model transformation in the GME is to write a C++ program that accesses a

[2] The mapping can be executed after bootstraping.

specific API that can obtain information about a model instance through abstractions related to concepts from the GME meta-meta-model. ANEMIC provides an ability to write model interpreters in C++ that use an API that has been generated as a customized wrapper, where the interface that the programmer uses is specific to the concepts in the domain and are mapped down to the native GME API calls. Although this does help to write GME model interpreters in a more domain-appropriate manner, ANEMIC generates C++ code rather than a more traditional model transformation language.

6 Conclusion

In this chapter, we introduced a language engineering technique for building MTLs that is based on treating each MTL as a domain-specific language, more specifically, as transformation-specific languages. In this approach, all the components of an MTL are modeled explicitly at the proper level of abstraction using the most appropriate formalism. MoTif was used as an example to illustrate the approach. Being a completely modeled language both at the syntax and at the semantics level, the MoTif language allows one to easily design HOTs. In fact, the semantic mapping of MoTif to its semantic domain is itself expressed in MoTif as a HOT from the former to the latter.

The rationale behind domain-specific languages applies to the design of transformation languages. The approach proposed in this chapter is an enabler for the rapid and rigorous design and implementation of "custom" MTLs. On the one hand, such languages reduce the "semantic gap" between the language and the application domain. On the other hand, if those languages are sufficiently restricted (e.g., non-Turing complete [3]), transformations modeled in them become amenable to formal analysis which is computationally infeasible for general-purpose MTLs. In this chapter, we do not give practical examples of domain-specific MTLs. Thus, our approach is only recently gaining acceptance and hence very little empirical evidence is available yet.

We are currently investigating how to model the pragmatics of MTLs. The pragmatics of T-Core were defined in natural language and UML sequence diagrams. In the future, we will investigate the representation of design patterns for MoTif as well as anti-patterns to model the pragmatics of the language explicitly.

Acknowledgements This work is supported partially by NSF CAREER award CCF-1052616.

References

1. Baar, T., Whittle, J.: On the usage of concrete syntax in model transformation rules. In: Virbitskaite, I., Voronkov, A. (eds.) Perspectives of Systems Informatics, LNCS, vol. 4378, pp. 84–97. Springer, Novosibirsk (2007)

2. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: Symposium on Applied Computing, pp. 1280–1287. ACM, Dijon (2006)
3. Barroca, B., Lúcio, L., Amaral, V., Felix, R., Sousa, V.: DSLTrans: A turing incomplete transformation language. In: International Conference on Software Language Engineering, LNCS. Springer, Eindhoven (2010)
4. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? Transformation models! In: Model Driven Engineering Languages and Systems, LNCS, vol. 4199, pp. 440–453. Springer, Genova (2006)
5. Bolduc, J.S., Vangheluwe, H.: The modelling and simulation package pythonDEVS for classical hierarchical DEVS. MSDL technical report msdl-tr-2001–01, McGill University (2001)
6. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. Spec. Issue Model Driven Software Dev. **45**(3), 621–645 (2006)
7. Dong, X.: ARK, the metamodelling kernel for domain specific modelling. Master's thesis, McGill University (2011)
8. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1: Foundations. World Scientific, Singapore (1997)
9. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object oriented and rule-based design of visual languages using Tiger. In: Zündorf, A., Varró, D. (eds.) International Workshop on Graph Based Tools, ECEASST, vol. 1, pp. 1–13. Natal (2006)
10. France, R., Ghosh, S., Dinh Trong, T., Solberg, A.: Model-driven development using UML 2.0: Promises and pitfalls. IEEE Comput. **39**(2), 59–66 (2006)
11. Gray, J., Tolvanen, J.P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-specific modeling. In: Fishwick, P.A. (ed.) CRC Handbook of Dynamic System Modeling, Chap. 7, pp. 1–20. CRC (2007)
12. Object Management Group: Meta Object Facility 2.4 Core Specification (2011). http://www.omg.org/spec/MOF/. Access 24 Apr 2013
13. Object Management Group: Meta Object Facility 2.0 Query/View/Transformation Specification (2011). http://www.omg.org/spec/QVT/. Access 24 Apr 2013
14. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "Semantics"? Computer **37**(10), 64–72 (2004)
15. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach. In: International Conference on Software Engineering, pp. 105–115. ACM, Orlando (2002)
16. 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.) International Conference on Graph Transformation, LNCS, vol. 2505, pp. 161–176. Springer, Barcelona (2002)
17. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
18. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In: Iivari, J., Lyytinen, K., Rossi, M. (eds.) Conference on Advanced Information Systems Engineering, LNCS, vol. 1080, pp. 1–21. Springer, Crete (1996)
19. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, Hoboken (2008)
20. Kilov, H.: From semantic to object-oriented data modeling. In: First International Conference on System Integration, pp. 385–393. IEEE, Piscataway (1990)
21. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained. The Model Driven Architecture: Practice And Promise. Reading, Addison-Wesley, Boston (2003)
22. Kühne, T.: Matters of (meta-)modeling. J. Softw. Syst. Model. **5**(4), 369–385 (2006)
23. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Systematic transformation development. Electron. Comm. Eur. Assoc. Software Sci. Tech. **21** (2009)

24. Lano, K., Kolahdouz Rahimi, S., Poernomo, I.: Comparative evaluation of model transformation specification approaches. Int. J. Softw. Informat. **6**(2), 233–269 (2012)
25. de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. In: Kutsche, R.D., Weber, H. (eds.) Fundamental Approaches to Software Engineering, LNCS, vol. 2306, pp. 174–188. Springer, Grenoble (2002)
26. de Lara, J., Vangheluwe, H., Moreno, M.A.: Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. J. Softw. Syst. Model. **3**(3), 194–209 (2004)
27. Lédeczi, Á., Bakay, A., Maroti, M., V"olgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE Comput. **34**(11), 44–51 (2001)
28. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model transformation with a visual control flow language. Int. J. Comput. Sci. **1**(1), 45–53 (2006)
29. Mosterman, P.J., Vangheluwe, H.: Computer automated multi-paradigm modeling: an introduction. Simulat. Trans. Soc. Model. Simulat. Int. **80**(9), 433–450 (2004)
30. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Briand, L., Kent, S. (eds.) MODELS/UML'2005, LNCS, vol. 3713, pp. 264–278. Springer, Montego Bay (2005)
31. Nordstrom, S., Shetty, S., Chhokra, K.G., Sprinkle, J., Eames, B., Lédeczi, Á.: ANEMIC: automatic interface enabler for model integrated computing. In: Generative Programming and Component Engineering, LNCS, vol. 2830, pp. 138–150. Springer, New York (2003)
32. Rumpe, B., Weisemöller, I.: A domain specific transformation language. In: Models and Evolution. Wellington (2011). Available at: http://www.se-rwth.de/publications/BR.IW.A.Domain.Specific.Transformation.Language_ME_2011.pdf
33. Sendall, S., Kozaczynski, W.: Model Transformation: The heart and soul of model-driven software development. IEEE Softw. **20**, 42–45 (2003)
34. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development – Technology, Engineering, Management. Wiley, West Sussex (2006)
35. Syriani, E.: A multi-paradigm foundation for model transformation language engineering. Ph.d. thesis, McGill University (2011)
36. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with DEVS. In: Nagl, M., Schürr, A. (eds.) International Symposium on the Applications of Graph Transformations with Industrial Relevance, LNCS, vol. 5088, pp. 136–152. Springer, Kassel (2007)
37. Syriani, E., Vangheluwe, H.: De-/Re-constructing model transformation languages. Electron. Comm. Eur. Assoc. Software Sci. Tech. **29** (2010) Available at: http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/407
38. Syriani, E., Vangheluwe, H.: A modular timed model transformation language. J. Softw. Syst. Model. **11**, 1–28 (2011)
39. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Paige, R., Hartman, A., Rensink, A. (eds.) European Conference on Model Driven Architecture: Foundations and Applications, LNCS, vol. 5562, pp. 18–33. Springer, Enschede (2009)
40. Wimmer, M., Kusel, A., Schönböck, J., Reiter, T., Retschitzegger, W., Schwinger, W.: Lets's play the token game – model transformations powered by transformation nets. In: Workshop on Petri Nets and Software Engineering, pp. 35–50. Université Paris 13, Paris (2009)
41. Zeigler, B.P.: Multifacetted Modelling and Discrete Event Simulation. Academic Press, San Diego (1984)

A Reconciliation Framework to Support Cooperative Work with DSM

Amanuel Alemayehu Koshima, Vincent Englebert, and Philippe Thiran

Abstract Despite the fact that domain specific models (DSM) tools become very powerful and more frequently used, the support for their cooperation has not reached its full strength and the demand for model management is growing. In cooperative work, the decision agents are semi-autonomous and therefore a solution for reconciliating DSM after a concurrent evolution is needed. Computer supported cooperative work (CSCW) community proposes tools or techniques to ensure collaboration among general purpose modeling languages, but they do not usually give functionalities to support reconciliation and merging for asynchronous modifications. In addition, management of communications among members of a working group could also help to facilitate their collaboration. In this chapter, we propose a communication framework called DiCoMEF to manage exchanges of concurrently edited DSM models among a group of engineers. Besides, we present a reconciliation framework to merge concurrently evolved DSM models along with their metamodels.

Keywords Collaborative modeling • CSCW • DSML • EMF • Migration

1 Introduction

Domain specific modeling (DSM) languages have matured and been becoming powerful over the past few years and are now used as an efficient alternative to general purpose modeling languages (e.g., UML, Petri Nets) for modeling complex systems [17]. For instance, DSM are adopted by the model-driven engineering approach as a way to define the structure, behavior, and requirements of software

---

A.A. Koshima (✉) · V. Englebert · P. Thiran
PReCISE Research Center, University of Namur, Belgium
e-mail: amanuel.koshima@unamur.be; vincent.englebert@unamur.be;
philippe.thiran@unamur.be

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_10, © Springer-Verlag Berlin Heidelberg 2013

applications in specific domains. The main idea of DSM is to describe a solution directly by using domain concepts rather than generic modeling languages [36]. The benefits of this approach have been described in [18]. Domain specific modeling language uses models, metamodels and meta-metamodels to describe concepts at different abstraction levels. A model is an abstraction of a software system. A metamodel is a DSL oriented towards the representation of software development methodologies and endeavors [11]. Likewise, as models are described by metamodels, metamodels are also described by a meta-metamodel (i.e., MOF [28], MetaL [4], EMF/ECore [39]) that denotes a minimum set of concepts which define the languages (including generally itself).

DSM requires ad hoc environment tools, called metaCASE, that enable method engineers to edit and manage models as well as metamodels. Since 1990s, several tools have been developed such as Atom3 [23], GME [24], MetaDone [9], or MetaEdit+ [17]. However, most of these metaCASE tools consider the modeling process as a single user task [4]. This hypothesis is too restrictive with regard to how projects are managed. Modeling of software systems usually requires collaboration among members of a group with different scope and skills (i.e., middleware engineers, human interface designers, database experts, and business analysts). Hence, there is a need for group members to share modeling artifacts (i.e., model and metamodels) and synchronize their activities. Shared modeling artifacts could be edited and evolve concurrently throughout the development life cycle of a software application by different users. As a result, they might not seamlessly work together or the final result may not be what users want. In other words, modeling artifacts become inconsistent with each other.

DSM is tailored to a specific application domain so that it has to evolve in order to meet new requirements of stakeholders [41]. DSMs evolve by modifying their metamodel in order to satisfy new requirements [13]. Indeed, like other software artifacts, metamodels could also evolve throughout software development life cycles (i.e., analysis, design, testing, and maintenance) as a result of a better understanding of the problem domain or error corrections [12]. Because of metamodel evolution, the existing models might not conform to the new version of their metamodel. Therefore, these models need to be co-evolved with their respective metamodel so as to keep conformance. Hence, metaCASE tools need to support the evolution of metamodels, the co-evolution of models, and the reconciliation and merging of concurrently edited models and metamodels during the project life cycle.

Inconsistency of a shared work (i.e., models, and metamodels) is one of the main challenges that hinder cooperative work. Hence, conflicts that cause inconsistencies need to be identified and resolved: they could be *textual, syntactic, or semantic* conflicts [26]. The most commonly adopted approach to ensure collaboration is a central repository with merge mechanisms and lock techniques [27]. Unfortunately, locking technique is inadequate for a large number of users who work in parallel [1, 26]. Besides, in practice, this technique takes much time for users to resolve conflicts [1, 29]. In addition, this approach restricts users to be dependent on one repository. For example, tools such as MetaEdit+ [17] and EMFStore [19] provide this mode of collaboration.

Other modes of collaboration could consist in a group of people concerned by a cooperative task that is large, transient, not stable or even nondeterministic [37]. Besides, the interaction pattern among members of a group could be dynamic and users are semi-autonomous in their partial work. This type of collaboration allows each member to have his/her own copy of a shared work (i.e., (meta)model) and carry on his/her activity in isolation with other users or a central authority. A user later communicates his/her work by sending messages to other members [27]. Implementing the exchange of method chunks [31] could serve as a basis for this mode of collaboration. In this chapter, we consider a chunk as a cohesive and autonomous model. Standard formats like GXL [15], PNML [30], or XMI [40] were designed to facilitate the exchange of models among users. Even though these tools define quite well the structure of data model, they vary in their semantics. This results in a problem of interoperability among different CASE tools. But, it is still possible to exchange models among the same family of CASE tools. This work considers a (meta)model exchange between CASE tools of the same family. This mode of collaboration gives users a better control over their data and addresses the problem of being dependent on a single repository. But, it is challenging to keep all copies of modeling artifacts consistent because they could be modified concurrently by users.

Managing communication among members, detecting conflicts and reconciliating conflicting modifications could ensure collaboration among DSM tools. In this chapter, we propose a distributed collaborative model editing framework called DiCoMEF to ensure collaboration among DSM tools [22]. In DiCoMEF, every member of a group has his/her own local copy of shared work (i.e., (meta)model). Members communicate their activities by exchanging messages. Specifically, they exchange sequences of elementary change operations (i.e., create, delete, update) that are used to adapt models and metamodels. The adaptation of models and metamodels is captured by the history metamodel, which enriches change operations with enough information to transfer activities from one node to the other. Moreover, modifications of models and metamodels are controlled by human actors rather than by software agents. DiCoMEF uses EMF/Ecore [39] as its meta-metamodel definition.

This chapter is organized as follows: Sect. 2 describes collaborative model editing. Section 3 gives a short overview about eclipse modeling framework. Section 4 describes the architecture, communication, reconciliation framework of DiCoMEF and the model migration. Section 5 summarizes the related works. Finally, Sect. 6 describes the future work and conclusion.

2 Collaborative Modeling

Cooperative work is attributed to mutual interdependence of tasks among multiple users to produce specific products or services [37]. *Computer Supported Cooperative Work (CSCW)* is a cooperative work that employs computer systems to support

mutually interdependent works. Following Schmidt et al. [37], CSCW is a broad discipline that deals with how people work in groups and the underlying technological support. *Groupware* is a software system that is designed to support cooperative work. *Collaborative model editing* is a groupware in which computer system are used to ease users communication and reconciliation work of concurrently edited (meta)models [22].

Cooperative work is inherently distributed, meaning that tasks are allocated among members of a group [2]. Since multiple users are engaged in cooperative work, interactions among members and applications could be unpredictable: members could have their own goals, strategies, and experience levels that might lead to chaotic environment. So, policies (guidelines) are required to define and restrict mode of interactions among members and between members and applications so as to avoid conflicts and confusions [8]. Moreover, assignment of access control rights (roles) to members of cooperative ensembles helps to reduce chaos and improve coordination. Uncontrolled communication lets every member of a group propagates his/her local activity (i.e., modification of metamodel) to other members directly. This could cause a continuous discussion among members to solve conflicting proposals and it may even hamper collaborative work. On the contrary, a controlled communication manages activities of members (i.e., modifications). Specifically, a controller is assigned to supervise changes. Management of changes could be more efficient and effective if a controller has a knowledge of business domain and has a good modeling experience. Besides, s/he has authority to accept or reject modifications.

Reconciliation is a process that constitutes activities such as detection of conflicting modifications and meshing so as to merge concurrently edited (meta)models into a new version. A reconciliation process can be a priori or a posteriori [3]. In a priori reconciliation mechanism, members agree on common terms and communication protocols beforehand to avoid confusion and disorder. But, it is practically impossible to anticipate all contingencies in advance so that the posteriori reconciliation is also required to deal with such incidents in future. In order to merge conflicting versions, differences between two versions of (meta)models need to be identified, conflicts among two versions should be detected and resolved. Differences between two versions of (meta)models are derived using either *state-based comparison* or *change-based comparison* [1, 20, 26].

State-based comparison compares states of two versions of (meta)models with a common ancestor as input and derive their differences. This process is commonly referred to as differencing and it is computationally expensive [20]. *Change-based comparison* keeps track of changes whenever they occur and stores them into a repository. As a consequence, there is no need to calculate deltas (i.e., differences) later. *Operation-based comparison* is a special type of *change-based comparison* where deltas are represented as a sequence of change-operations [1, 20, 26]. *Operation-based comparison* captures the exact time sequences of changes that could help to understand changes and detect conflicts [1, 20, 26]. Besides, it can also express sets of operations that occurred in a common context as composite operations (i.e. refactoring operations). According to Koegel et al. [20], time

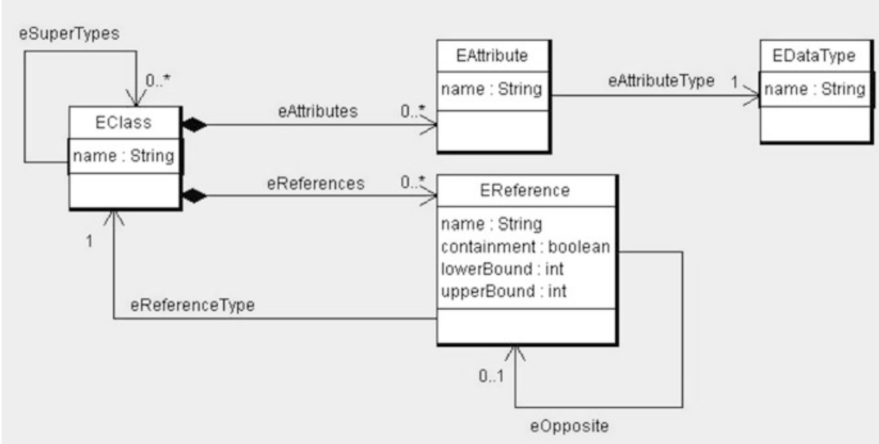


Fig. 1 A simplified subset of the Ecore metamodel

sequences of changes and composite operations help users to easily understand changes in *operation-based comparison* than in *state-based comparison*.

3 Eclipse Modeling Framework

This research is carried out in the context of the eclipse modeling framework (EMF). EMF is one of the most widely used modeling framework to build tools and applications. EMF generates codes (i.e., classes for the metamodel[1] and adapter classes for viewing and editing models) based on the structured data model [39]. A model could be expressed using annotated Java interfaces, XML Schema, or UML modeling tools. EMF provides a facility to generate one form of representation from the other (using the EMF framework). EMF uses Ecore as a meta-metamodel to define different DSL languages and itself.

The subset of Ecore model [39] is depicted in Fig. 1 that describes the Ecore metamodel. In EMF, a model class is represented by using an EClass, which is identified by a name and has zero or more attributes and references. A class can have zero or more super types. Although it is not depicted in the diagram, an EClass can have zero or more operations. Properties (attributes) of a class are modeled using an EAttribute, which has a name and a type. Associations are modeled by EReference(s). An EReference models an end of an association between two classes; it has a name and a type (the EClass at the opposite end of the association). A bi-directional navigable association is modeled using two references

[1]Ecore model.

that are related to each other by eOpposite link. Besides, a composition association is represented by setting a containment boolean property of an EReference to `true`. The cardinality of a reference is modeled by setting lowerBound and upperBound values. Like references, an attribute's cardinality could be specified using lowerBound and upperBound features. There are more model elements which are not covered in this chapter and we invite interested readers to refer to [39].

4 DiCoMEF

DiCoMEF is a distributed collaborative model editing framework where each member of a group has his/her own local copy of a (meta)model. DiCoMEF provides three kinds of membership for a collaborative group: *controllers* who manages (meta)model evolution; *editors* who have a read/write access on their artifacts; and *observers* who have only read access. These membership types are discussed in detail later in this chapter. Besides, members of a collaborative group communicate their work to other members by exchanging messages. The proposed architecture, communication, and reconciliation framework are presented in the following subsections.

4.1 Architecture

Figure 2 describes a collaboration scenario with the DiCoMEF framework with two groups of users, each one sharing a model—they each own a local copy of the artifact. In each group, one unique controller is responsible for the consistency of the model, hence, s/he owns the master copy.

Editors of Group 1 (Edward and Eric) and editors of Group 2 (Ephraim, Eden and Evan) have their own local copies. They can even modify/edit them. Afterwards, they have to send their modifications as a change request, respectively, to Group 1 controller (Caroline) and the Group 2 controller (Catherine) so as to propagate their modifications to other members. The controller supervises the change requests and propagates accepted changes to other members. As roles are not exclusive, a member of one group is allowed to be involved in one or more other groups. For example, an observer of one group might be a controller in another group as shown in Fig. 2. We assume that (meta)models owned by one controller is independent from (meta)models owned by other controllers meaning that there is no any (meta)model element which has a reference to (meta)model elements owned by other controllers. Moreover, only (meta)model are considered in this chapter, not source codes. DiCoMEF allows a member to exchange his/her local modification directly without supervision of a controller, but this type of communication has a risk to become out of synchrony with other members.

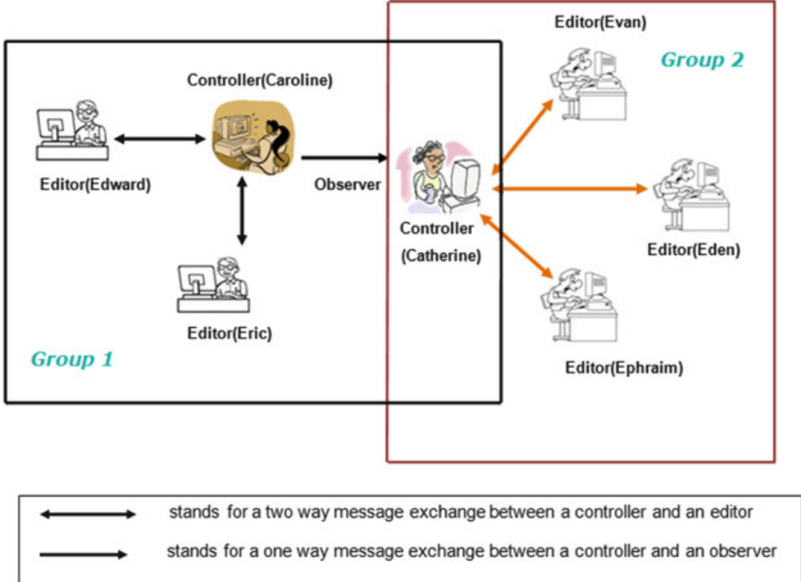


Fig. 2 Architecture of DiCoMEF

DiCoMEF metamodel expresses the main concepts used in DiCoMEF (see Fig. 3). These concepts are *person, role, role type, model, metamodel, copy model, and master model*. A master (meta)model is the main (meta)model which has one or more copy (meta)models that are distributed among editors and observers. DiCoMEF uses a universal unique identifier (UUID) to differentiate (meta)model elements (i.e., classes, attributes, references) uniquely. Two (meta)model elements are considered as identical if and only if they have the same UUID. Besides, a person involved in collaborative modeling has a role, which is typed as a *controller, editor*, or *observer*. In fact, there are two controller role types which are implemented in DiCoMEF such as a model controller or a metamodel controller. A metamodel control and a model-controller manages the evolution of a master metamodel, respectively, a master model. A controller role type is flexible meaning that it can be assigned (delegated) to other members of a group as long as there is one unique coordinator per group. A person who has an editor role can write and read his/her local copy (meta)models, whereas an observer role only has a read access to a local copy (meta)models.

Main-line and *branches* are the two important concepts that DiCoMEF relies on to store models and metamodels and ensures communication framework (see Fig. 4).[2] The main-line stores different versions of a copy (meta)model locally at

[2]Although these terms are also used by SCM programs, our framework does not rely on a central SCM.

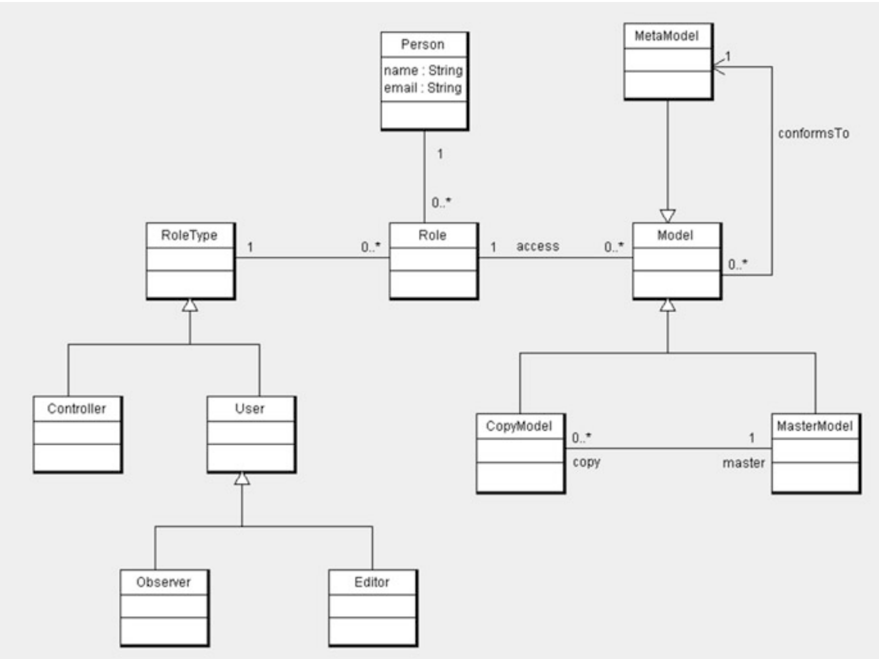


Fig. 3 DiCoMEF metamodel

each editors site. An editor cannot modify copy (meta)models stored on the main-line. Whenever s/he wants to modify copy (meta)models locally, s/he first creates a branch from the main-line and does modifications there. Afterwards, s/he sends her/his local modifications to a controller so as to commit those changes on the main-line. For example, in Fig. 4, a copy (meta)model evolves from version V_0 to version V_1 on the main-line based on changes propagated from a controller. It also shows a branch that is created by an editor to modify a copy (meta)model locally from version V_0 to version $V_{0.1}$; a branch was created before a copy (meta)model evolves from version V_0 to version V_1.

We use an Entity-Relationship model (ER-M) and metamodel (ER-MM) to demonstrate a collaboration scenario among a group of modelers and meta-modelers. Suppose Caroline is metamodel controller of a group that has two editors—Edward and Eric, whereas Miheret is a controller for a modeler group which constitutes two editors such as Elvis and Eyan. Caroline distributes the first version of the ER-MM (see Fig. 5) to editors Edward and Eric. Afterwards, Edward creates a branch from the main-line and extends the metamodel by adding a persistent field (*techno:String*) to the entity meta EClass and he renames the entity meta EClass to *"EntityType."* He could also annotate rationale of his modification using multimedia files (see Fig. 6). Later, he sends his local modifications back to Caroline so as to share his modifications with other members. Eric could also create a branch on his local machine from the first version of the ER-MM metamodel and

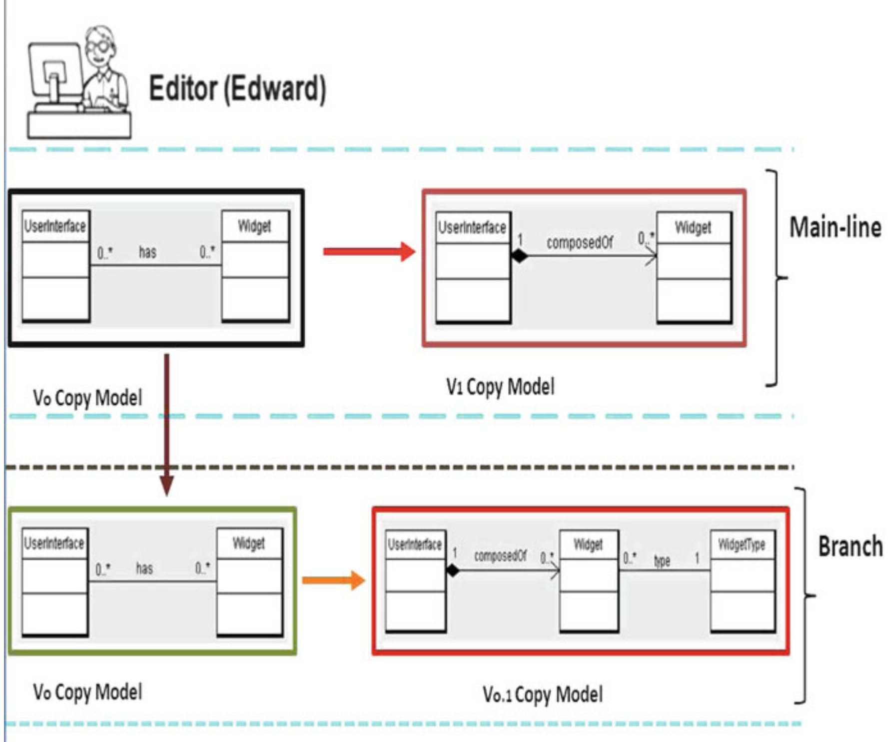


Fig. 4 Main-line and branch

modifies it by renaming the entity meta Eclass to *"Entity."* Suppose that Caroline accepts Bob's modification and propagates as accepted change for all members (i.e., Edward and Eric). These changes could result in conflicts with Eric's local modification (i.e., an entity meta EClass is assigned different names—*"Entity"* and *"EntityType"*). Like metamodelers group, modelers could have conflicting modifications. We will discuss the communication and reconciliation of conflicting modifications in the following section.

4.2 Communication

The communication framework of DiCoMEF is organized around the controller that acts as a central hub. When members of a group modify (meta)models locally, elementary change operations (i.e., create, delete and updates) are stored locally in a local repository. These elementary operations constitute a history that is used to propagate local modifications to the controller and secondarily to other members. Histories are defined by a history metamodel. Some works in the past have already used history metamodels. In [10], authors have developed a history metamodel

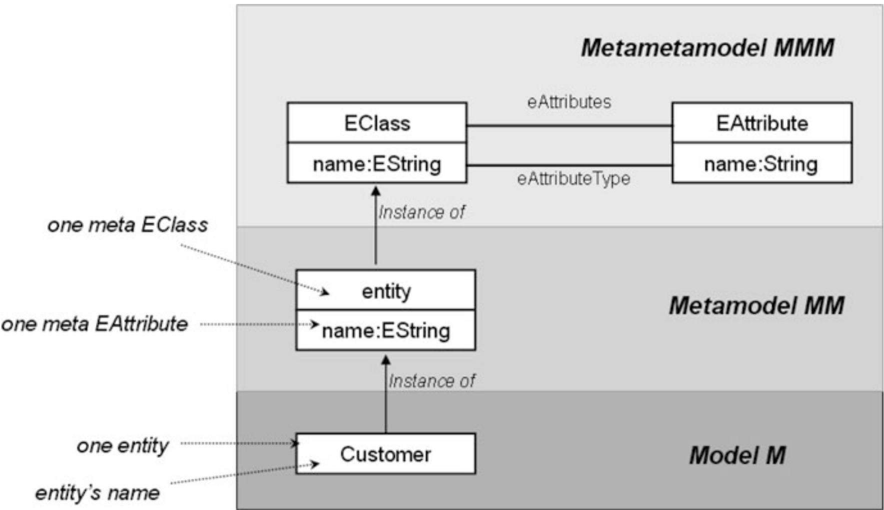


Fig. 5 The entity-relationship model M and its metamodel MM. The entity meta EClass owns one meta EAttribute (name) whose type is string. Customer is an entity

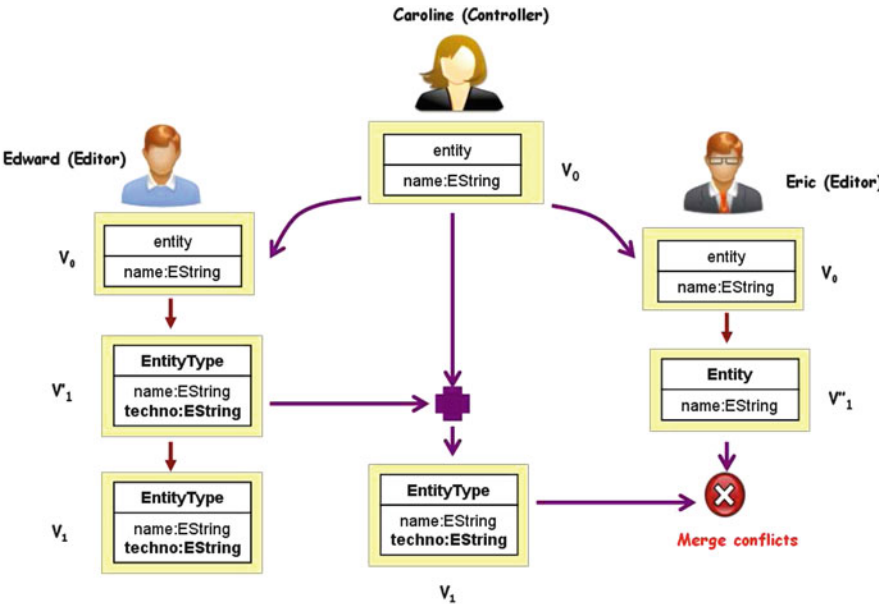


Fig. 6 Cooperative scenario. This figure shows the exchange of information and the successive evolutions of the (meta)models. Modifications are indicated with *bold letters*. *Cross symbols* denote import and reconciliation processes

called Hismo based on a FAMIX metamodel [5]. Hismo transforms a snapshot model like UML or FAMIX into a history instead of recording elementary edit operations while they occur. This type of transformation does not preserve time sequences of operation that could be useful for reconciliation process. EDAPT [13], previously called COPE, is a tool based on EMF/Ecore metamodel that captures edit operations of metamodel adaptation whenever they occur. But, in EDAPT, the history and the metamodel elements are highly coupled (i.e., each primitive operations like create, set and add are linked with a model element via elements reference). As a result, it is impossible to send the history (i.e., adaptation operations) without shipping together the metamodel elements that are adapted. Besides, EDAPT does not provide a facility to define compound operations from other compound operations, which could give users more flexibility on how to organize the history in a hierarchic way.

Like EDAPT, EMFStore [19] is developed based on the EMF/Ecore metamodel; it captures histories of metamodel evolution but the history and the metamodel elements are not coupled meaning that there is no any direct reference or link from the history elements (i.e., set, add, create operations) to metamodel elements. But, by the time this research was conducted, the history metamodel of EMFStore was tightly coupled with other components of the EMFStore implementation. As a result, EMFStore cannot be used/installed as an autonomous component for capturing history of metamodel adaptation.[3] Therefore, we developed a tool that captures a history of metamodel adaptation by extending EDAPT (see Fig. 7). The extension is inspired by the EMFStore history metamodel [19] that eliminates coupling between metamodel elements and history elements. As shown in Fig. 7, a primitive operation refers to a model element using a model element Id rather than directly pointing out a model element. This avoids coupling between model elements and history elements. Besides, a create and delete history elements have direct containment references to a model element that are newly created and deleted, respectively. This reference is used to create or populate a model element in other nodes and it can also be used to identify a deleted model and its child element(s).

DiCoMEF implements two distinct history metamodels: a version for metamodel adaptation (see Fig. 7) and another one for model adaptation. Both follow the same goal but they, nevertheless, exhibit some differences. The main difference comes from the semantics of multivalued attributes and references. They are considered as sets at the metamodel level (the order of element does not matter) although they are considered as lists at the model level. Hence, the history metamodel for metamodel adaptation does not have an index field for operations like *Add, Remove, Move*. The other difference is with the *Move* operation. For metamodel adaptation, the source and the target metamodel element should be different, whereas the source and the target model element should be the same in the case of model adaptation. Hence,

[3] Recently, EMFStore has had refactoring to reduce a coupling between parts of the implementation that captures history with rest of implementation.

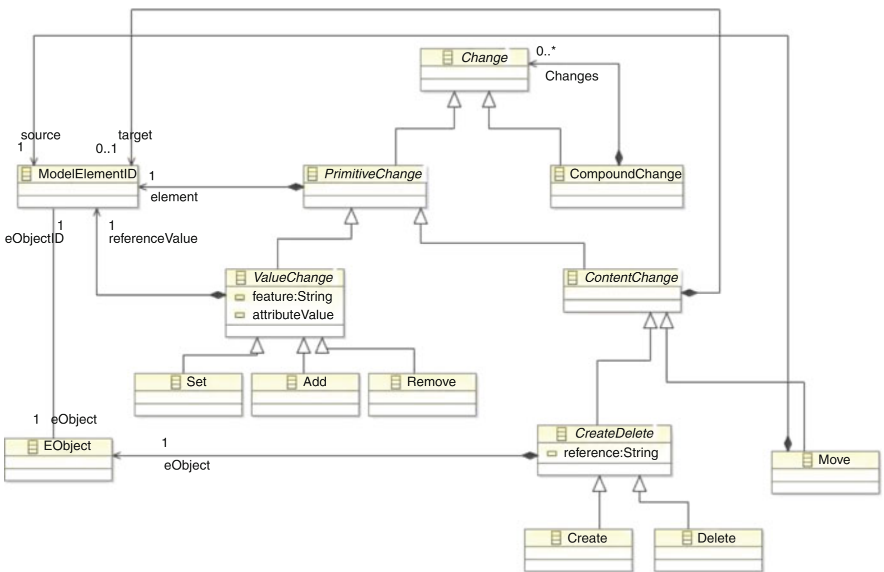


Fig. 7 The history metamodel

these two references are merged into one reference. DiCoMEF history metamodels are self-contained: links and references are replaced by a surrogate mechanism. As a result, one can exchange the history without (meta)model elements.

In DiCoMEF, editors can use multimedia files (i.e., video, audio, text) to annotate change operations with the rationale of their intent in order facilitate the reconciliation process later. After editing (meta)models locally and annotating change operations, editors send their change operations as a change request to a controller so as to share their local modifications with other members. This work assumes that modifications of (meta)model are supervised by a controller who has a good modeling experience and a good knowledge of the business domain. We also believe that giving him the authority to accept or reject modifications could benefit collaborative work, specifically, in a deadlock situation where editors do not agree on modifications. The controller inspects proposed changes by an editor and applies accepted changes on a master (meta)model (when it evolves from version V_n to V_{n+1}). Indeed, s/he can consult multimedia files for a better understanding of proposed changes or s/he can directly contact an editor who proposes changes; the controller and editor could elaborate the (meta)model together. Eventually, the controller propagates accepted changes to all members of a group so as to automatically evolve copy (meta)models (i.e., stored on main-lines) from version V_n to version V_{n+1}. These modifications could lead to a new version of a copy (meta)model that is inconsistent with local modifications. Hence, these conflicts should be identified and reconciled locally by editors.

While a model controller communicates only the model adaptations to his/her users group, metamodel controllers have to inform the model controllers of the model migration instructions. For example, as it is shown in Fig. 6, Edward modifies the ER-MM metamodel and sends the sequence of modification operations (i.e., annotated operations) to the controller Caroline for examination—they can work together on that task. Afterwards, she propagates the changes to all the members (i.e., metamodelers) so as to evolve copy metamodels from version V_0 to version V_1. She can also send a model migration strategy for a model controller (i.e., Miheret) to evolve models to be conformed with new definitions of metamodel. Model migration will be discussed in detail later in this chapter. Miheret (model controller) migrates instance models based on migration instructions and distributes migration instructions to modelers such as Elvis and Eyan. Besides, Miheret also manages which version of a metamodel editor is used in a modeler group.

4.3 Reconciliation

Our framework allows people to work on models in an asynchronous way and to exchange their versions. This convenience can lead to conflicts that must be detected and resolved during a *reconciliation task*. This includes activities like model comparison, conflict detection, and reconciliation. DiCoMEF adopts an *operation-based model comparison* to derive differences between two (meta)models. Every modification operation a user makes is captured and stored. This information is modeled by a history metamodel (see Fig. 7) and used next to compute the delta between two successive versions of a (meta)model. DiCoMEF detects conflicts by inspecting a list of sequences of elementary operations that have modified a (meta)model [1, 21, 25, 26]. This approach could be regarded as a operation-based conflict detection process [21]. Moreover, users can explicit the rationale of modifications with multimedia files (i.e., text, images, images, audio), which could be useful to resolve conflicts.

Reconciliation is done for pairs of changes. When they are not conflicting, no reconciliation is required. This happens, for instance, if a pair of changes modifies an element in the same way: both Edward and Eric rename a meta EClass as “EntityType.” On the other hand, renaming a meta EClass differently (see Fig. 6) must be considered as a conflict. In that case, the controller and editors consult multimedia files so as to better understand rationale of modifications. Editors can also contact (i.e., via video conferences, email, chat) a controller or another editor who proposed changes in order to solve conflicts together.

An editor and a controller can also elaborate a change request together. Suppose Caroline (controller) sends a first version of a model to Edward (editor). Edward modifies the model and sends a change request to Caroline (see Fig. 8b). Caroline examines proposed changes and suggests to add a new class called a *WidgetType* because a same widget type (i.e., Button, TextField, Label) might be used by different user interfaces. Caroline and Edward elaborate the model together by

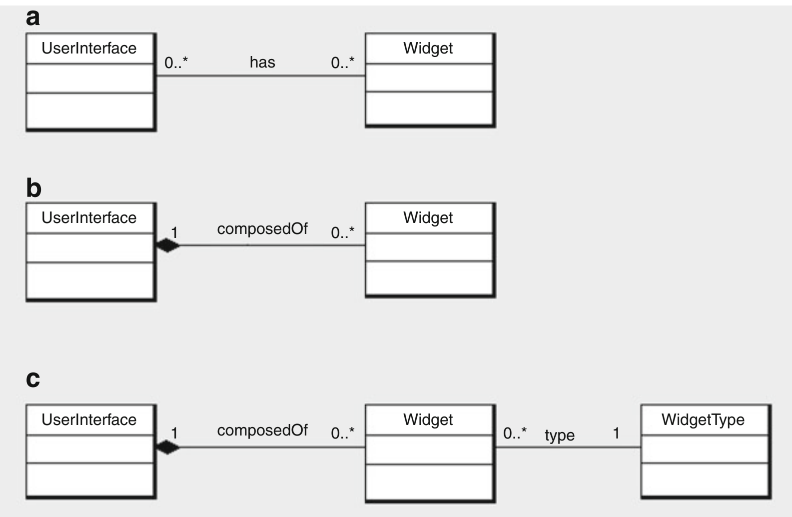


Fig. 8 Cooperative scenario: a controller and an editor elaborate model together

adding a class *(WidgetType)* and annotate change operations with rationale of modification (see Fig. 8c). Later, Caroline sends changes that she and Edward agreed up on as a change propagation to other members. As a general rule of reconciliation, this work assumes that every editor must apply modifications proposed by a controller whenever conflicts occur. But, an editor can propose her/his local conflicting modifications as a change request later. For example, Eric renames a meta EClass to *"EntityType"* first and he could send his modification later to Caroline (see Fig. 6).

As it was discussed above in Sect. 4.1, newly created model elements are assigned a new identifying UUID value. As a result, they are considered as different whatever their characteristics are. Hence, if Edward and Eric create each one a persistence attribute called *techno:String* in their respective local ER-MM metamodel (see Fig. 6), they would be considered as distinct even if they represent conceptually the same model element. Hence, these redundant model elements need to be manually merged into one model element. The merging process cannot be automatic since it could depend on the semantics that they are not aware of. As a general rule, if a same model element (i.e., conceptually the same) is created locally and in a change propagation (from a controller), a merged model element should be assigned the UUID value of a model element in a change propagation.

Editors synchronize their local modifications with propagated changes (from a controller) in two fashions. They either synchronize their local modifications with propagated changes first and continue with their work or delay the synchronization activity until they finish their work and do it later. In both cases, a copy (meta)model firstly evolves from version V_n to version V_{n+1}. Afterwards, operations that adapt a copy (meta)model V_n locally are re-played on a new version, V_{n+1}.

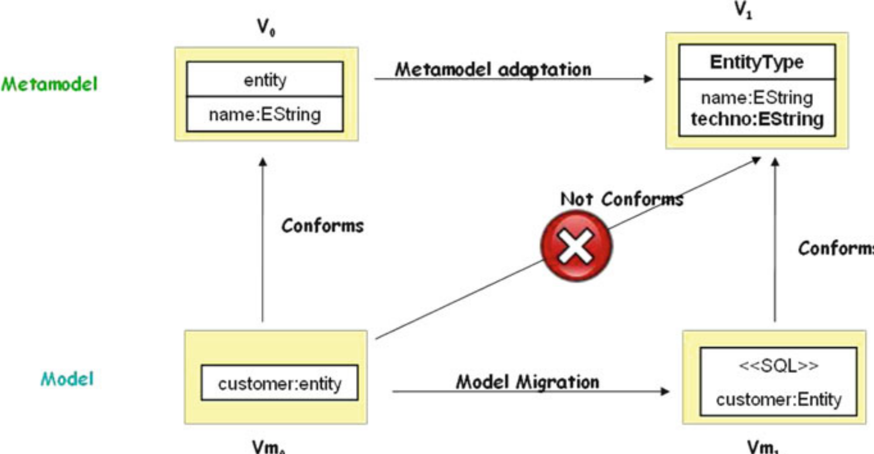


Fig. 9 Example of model migration

4.4 Model Migration

Like other software artifacts, metamodels may evolve to meet new requirements [13]. Hence, its instance models might not anymore satisfy the rules and constraints specified by the new version. This would even forbid these models to be edited with editors built for an old metamodel. Model migration consists in adapting models in response to their metamodel evolution so as to keep them conform with their new version of metamodels [14]. For example, as shown in (Fig. 9), a model version Vm_0 conforms with metamodel version V_0, but it violates constraints defined by metamodel version V_1. Therefore, a model version Vm_0 needs to be migrated to version Vm_1 so as to conform with evolved metamodel, Vm_1.

In [34], authors have classified model migration as *manual specification*, *operator-based approach*, and *metamodel matching approach*. Graph transformation languages could also be used to migrate models, but as shown in [32], they are not easy to understand and results of migration could be incorrect. *Metamodel matching approach* compares the source and target metamodels so as to derive a difference model and generate migration instructions. Nevertheless, this approach cannot always automatically generate correct migration instructions [34]. It is thus not a suitable choice for domains that require completeness, correctness, and predictability. The *Operation-based approach* uses the sequences of modification operations to derive a migration strategy at the model level. Indeed, operation-based approach needs an integration of a library of coupled operations with modeling tools. A coupled operation brings together a metamodel adaptation operator and a model migration strategy. This approach captures a user intention (i.e., keeps operations, which are contained in a composite operation, in the same context) while incrementally adapting metamodels [14].

Collaborative modeling frameworks, which exchange sequences of modification operations as a means of communication among members, mostly normalizes them so as to speed up the transfer and to reduce the complexity of the merging process. So, operations can be superseded by new ones and can thus be cleaned from the history. For example, a create operation will be removed if it is followed by a delete operation on the same object. Canonization could also be applied for a library of coupled-operations defined. A coupled operation could be composed of one or many primitive operations (i.e., create, delete, set, add, and remove) and model migration instructions. For example, in EDAPT [7, 13], a *Create Attribute* coupled-operation is composed of primitive operations that create an attribute and set values for name, type, minimum and maximum cardinality, and default value of a newly created attribute. For instance, if a *Create Attribute* coupled-operation is followed by a *Change Attribute Type* coupled-operation, a primitive operation that sets a type of an attribute in *Create Attribute* coupled-operation needs to be deleted due to canonization. As a result, the *Create Attribute* coupled-operation becomes invalid; the canonized set of primitive operations do not satisfy constraints that state a *Create Attribute* coupled-operation must set a type value of a newly created attribute. Hence, sets of canonized primitive operations that are composed by coupled-operations need to be re-grouped into new sets of valid coupled-operations. Primitive operations are re-grouped into a coupled-operation in order to use model migration instructions defined by the coupled-operation. For primitive operations that are not composed by coupled operations and make instance models inconsistent, a model migration instruction should be written manually. Manually incorporating model migration instructions is a tedious and error prone task.

Re-grouping primitive operations into library of coupled operations manually is a tedious and difficult task. Users need to know the structure and the constraints of each coupled operation so as to compose it from primitive operations. Canonization also makes re-grouping more complicated by removing some primitive operations. This could invalidate constraints of a coupled operation, for example, a *Create Attribute* coupled-operation becomes inconsistent if a *Set Attribute Type* operation is removed from it. Hence, model migration cannot be applied.

Manual model migration [34] approach needs model migration instructions to be specified manually. This approach does not require a library of coupled-operations and it can be defined in anyway permitted by modeling tools so as to migrate models. It gives a better control for users to manage model migration. But, manual specification needs more effort from a user to encode migration instructions and to integrate them with modeling tools. Besides, it does not use re-occurring patterns in model migration and it could be error prone.

Manual model migration can be augmented with a history of metamodel changes so as to write a correct and useful model migration instructions. The history can be used to correctly identify deleted, created, and moved model elements. This work adopts a manual model migration approaches and relies on Epsilon Flock [33]. It is a domain-specific language used to specify model migration instructions. Besides, it provides facilities to execute migration instructions. Epsilon Flock adopts

a manual model migration approach, but it reduces manual efforts by employing a conservative copying algorithm that automatically copies only model elements that conform to a new metamodel. Epsilon Flock performed better in terms of correctness, conciseness, understandability, extensions, and appropriateness than other model migration techniques that took part in transformation tool contest 2010 [32].

DiCoMEF uses the history of the metamodel adaptation to help user to encode a useful model migration instructions, which might reflect the user's intention. Besides, DiCoMEF keeps all metamodel elements and model elements time-aware. Each model version keeps information about a metamodel version(s) with whom it comply and it could also be migrated on demand to respect constraints defined by new metamodel.

It is also worth mentioning that due to model migration, a history of model adaptation becomes inconsistent with an instance model. For example, if an existing class is deleted from a metamodel, then instances of the same class type need to be deleted from the migrated model. As a result, history elements (i.e., change operations create, set and add) that manipulate instance model elements are referring to classifiers that don't exist anymore in the modeling language. Hence, that history also needs to be migrated along with its instance model. A model migration instruction used to migrate instance model can also be used to migrate a history as well. A change operations that refer to a classifier or a feature that does not exist should be removed.

5 Related Work

Many research has been already done to address challenges of collaborative software development. In [16], Ignat et al. compared different approaches for collaboratively editing a text or tree-based documents. Dewan and Hedge [6] also proposed a collaboration model that lets users handle conflicts and merge their intentions collaboratively. However, most of the previous work deal with collaborative merging of software codes.

Saeki [35] introduced the use of versioning system to control and manage models and metamodels, which evolve independently. The author did not consider collaboration in his work. In [4], Constantin et al. proposed a reconciliation framework for collaborative model editing. In their work, they suggested a weakly coupled mode of collaboration, where (meta)models are managed in distributed fashion. But, they only provide a theoretical reconciliation framework to support collaborative work without providing a solution. There are few frameworks available that support collaboration among DSML tools. These frameworks commonly adopt approaches like using central repository with merge mechanisms and locks [38] in order to ensure collaboration and handle inconsistency problems. EMFStore is an operation-based collaborative model editing framework for Eclipse Modeling Framework

(EMF)-based models [19]. EMFStore uses a central repository with copy-merge techniques to ensure collaboration. MetaEdit+ [17] implements *Smart Mode Access Restricting Technology* (Smart Locks ©) to support concurrent access of shared modeling artifacts that are stored centrally. Even though locking technique assumes strict consistency model, it becomes inadequate when a number of users who edit (meta)models in parallel reach a very low threshold [26]. Moreover, these approaches constrain all members to be dependent on a central repository. In [27], Mougenot et al. developed a peer-to-peer collaborative model editing framework called D-Praxis. It lets users exchanging sequences of operations used to adapt a model as a means of communication. This approach implemented automatic conflict resolution based on delete semantics and Lamport clock. Nevertheless, this approach suffers from similar problem of "lost-update." We argue that final results of automatic reconciliation process could not reflect the intention of users. So, we propose a distributed collaborative framework called DiCoMEF which free users being dependent on a central repository. Besides, modifications are controlled by human agents (not automatic).

6 Future Works and Conclusion

To fully benefit from DSM tools, it is important to improve cooperation among them. Approaches based on locking techniques and a central repository might not meet the new requirements of new groupwares. This chapter has presented a theoretical framework to ensure collaboration among DSM tools. Specifically, managing communications among members of a collaborative work and reconciliating concurrently evolved DSMs. DiCoMEF allows a group of modelers to share the same domain-specific model and to work in isolation. Modelers are organized as coordinators, editors, and observers. The framework allows them to distribute models as well as their metamodel to let them work concurrently. Modifications are managed by a controller (human agent). More importantly, a controller role is flexible meaning that it could be easily assigned to another member. This dynamic roles assignment could let people to implement more elaborated strategies on top of DiCoMEF, i.e., a user can delegate his/her role to another person. Although using a controller to manage collaborative modeling has a limitation of scalability, it could be possible to implement different method engineering techniques (e.g., delegation mechanisms, pooling) and strategies on top of DiCoMEF to address the problem. DiCoMEF also provides facilities for editors to annotate their rationale of changes with multimedia files. The proposed framework, DiCoMEF, is under implementation for validating theoretical concepts presented in this chapter.

References

1. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. Tech. rep., Johannes Kepler University Linz (2009)
2. Borghoff, U., Schlichter, J.: Computer Supported Cooperative Work: Introduction to Distributed Applications. Springer, Berlin/Heidelberg (2000)
3. Boukhebouze, M., Koshima, A., Thiran, P., Englebert, V.: Comparative analysis of collaborative approaches for UsiXML meta-models evolution. In: 1st International User Interface eXtensible Markup Language Workshop, EICS 2009 Conference, pp. 9–14. Thales Research and Technology France, France (2010)
4. Constantin, C., Englebert, V., Thiran, P.: A reconciliation framework to support cooperative work with DSM. In: Proceedings of the First International Workshop on Domain Engineering Held in Conjunction with CAiSE'09 Conference, collection CEUR-WS.org, vol. 457 (2009)
5. Demeyer, S., Tichelaar, S., Ducasse, S.: FAMIX 2.1- the FAMOOS information exchange model. Tech. rep., University of Bern (2001)
6. Dewan, P., Hegde, R.: Semi-synchronous conflict detection and resolution in asynchronous software development. In: Harper, R., Gutwin, C. (eds.) ECSCW, pp. 159–178. Springer, Berlin/Heidelberg (2007)
7. EDAPT: Framework for Ecore model adaptation and instance migration. http://www.eclipse.org/proposals/edapt/ (2012)
8. Edwards, W.K.: Policies and roles in collaborative applications. In: Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work, CSCW '96, pp. 11–20. ACM, New York, NY (1996)
9. Englebert, V., Heymans, P.: Towards more extensible meta-CASE tools. In: Krogstie, J., Opdhal, A., Sindre, G. (eds.) International Conference on Advanced Information Systems Engineering (CAiSE'07), no. 4495 in LNCS, pp. 454–468 (2007)
10. Gîrba, T., Favre, J.M., Ducasse, S.: Using meta-model transformation to model software evolution. Electron. Note Theor. Comput. Sci. **137**, 57–64 (2005). DOI http://dx.doi.org/10.1016/j.entcs.2005.07.005. URL http://dx.doi.org/10.1016/j.entcs.2005.07.005
11. Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley, New York (2008)
12. Gruschko, B.: Towards synchronizing models with evolving metamodels. In: Proc. Int. Workshop on Model-Driven Software Evolution Held with the ECSMR (2007)
13. Herrmannsdoerfer, M.: Operation-based versioning of metamodels with cope. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, CVSM '09, pp. 49–54. IEEE Computer Society, Washington, DC (2009). DOI http://dx.doi.org/10.1109/CVSM.2009.5071722. URL http://dx.doi.org/10.1109/CVSM.2009.5071722
14. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE: A Language for the Coupled Evolution of Metamodels and Models. In: Proc. of the 1st International Workshop on Model Co-Evolution and Consistency Management. ACM, New York, NY (2008)
15. 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)
16. Ignat, C.L., Oster, G., Molli, P., Cart, M., Ferrie, J., Kermarrec, A.M., Sutra, P., Shapiro, M., Benmouffok, L., Busca, J.M., Guerraoui, R.: A comparison of optimistic approaches to collaborative editing of Wiki pages. In: Proceedings of the 2007 International Conference on Collaborative Computing: Networking, Applications and Worksharing, pp. 474–483. IEEE Computer Society, Washington, DC, USA (2007). DOI 10.1109/COLCOM.2007.4553878
17. Kelly, S.: CASE tool support for co-operative work in information system design. In: Rolland, C., Chen, Y., Fang, M. (eds.) IFIP TC8/WG8.1 Working Conference on Information Systems in the WWW Environment, pp. 49–69. Chapman & Hall (1998)
18. Kelly, S., Tolvanen, J.P.: Domain-specific modeling enabling full code generation. Wiley-Interscience IEEE Computer Society, Hoboken (2008)

19. Koegel, M., Helming, J.: EMFStore: a model repository for emf models. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) ICSE (2), pp. 307–308. ACM, New York, NY (2010)
20. Koegel, M., Herrmannsdoerfer, M., Helming, J., Li, Y.: State-based vs. operation-based change tracking. In: Proceedings of MODELS'09 MoDSE-MCCM Workshop. Denver, USA (2009). URL http://wwwbruegge.in.tum.de/static/publications/pdf/205/Paper3.pdf
21. Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., Helming, J.: Operation-based conflict detection. In: Proceedings of the 1st International Workshop on Model Comparison in Practice, IWMCP '10, pp. 21–30. ACM, New York, NY (2010)
22. Koshima, A., Englebert, V., Thiran, P.: Distributed collaborative model editing framework for domain specific modeling tools. In: ICGSE, pp. 113–118. IEEE (2011)
23. de Lara, J., Vangheluwe, H.: Using AToM as a meta-CASE tool. In: ICEIS'02, pp. 642–649 (2002)
24. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing (2001)
25. Lippe, E., van Oosterom, N.: Operation-based merging. SIGSOFT Software Eng. Note **17**, 78–87 (1992)
26. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Software Eng. **28**, 449–462 (2002)
27. Mougenot, A., Blanc, X., Gervais, M.P.: D-Praxis: A peer-to-peer collaborative model editing framework. In: Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems, DAIS '09, pp. 16–29. Springer, Berlin/Heidelberg (2009)
28. Object Management Group (OMG): Meta Object Facility (MOF) Specification. http://www.omg.org/spec/MOF/1.4/PDF (2002)
29. Pilato, C., Collins-Sussman, B., Fitzpatrick, B.: Version Control with Subversion, 2nd edn. O'Reilly Media, Sebastopol (2008)
30. Petri net markup language PNML. http://www.pnml.org/ (2011)
31. Ralyté, J., Roll, C.: An approach for method reengineering. In: Proceedings of the 20 th International Conference on Conceptual Modeling (ER2001), LNCS 2224, pp. 471–484. Springer, Berlin/Heidelberg (2001)
32. Rose, L., Herrmannsdoerfer, M., Mazanek, S., Van Gorp, P., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schätz, B., Wimmer, M.: Graph and model transformation tools for model migration. Software and Systems Modeling pp. 1–37 (2012). URL http://dx.doi.org/10.1007/s10270-012-0245-0. 10.1007/s10270-012-0245-0
33. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model migration with epsilon flock. In: Proceedings of the Third international conference on Theory and practice of model transformations, ICMT'10, pp. 184–198. Springer, Berlin/Heidelberg (2010). URL http://dl.acm.org/citation.cfm?id=1875847.1875862
34. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: An analysis of approaches to model migration. In: Proc. Models and Evolution (MoDSE-MCCM) Workshop, 12th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems (2009)
35. Saeki, M.: Configuration management in a method engineering context. In: Dubois, E., Pohl, K. (eds.) CAiSE, Lecture Notes in Computer Science, vol. 4001, pp. 384–398. Springer, Berlin/Heidelberg (2006)
36. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Comput. **39**(2), 25–31 (2006)
37. Schmidt, K., Bannon, L.: Taking CSCW seriously: Supporting articulation work. Comput. Supported Cooper. Work **1**, 7–40 (1992)

38. Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting collaborative development in an open MDA environment. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance, pp. 244–253. IEEE Computer Society, Washington, DC (2006). DOI 10.1109/ICSM.2006.64
39. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Upper Saddle River (2009)
40. OMG, XMI mapping specification, v2.1.1, formal/07-12-0 (2007)
41. Zhang, J.: Metamodel-driven model interpreter evolution. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA ’05, pp. 214–215. ACM, New York, NY (2005)

Part III
Conceptual Modeling

Model Oriented Domain Analysis and Engineering

Jorn Bettin

Abstract Model oriented domain analysis and engineering (MODA & MODE) is a methodology for value chain analysis and domain engineering that can be used to uncover and formalize the knowledge that is inherent in any software intensive business or any scientific discipline. The target audience consists of domain experts in any line of business or field of scientific endeavour and is not limited to software development professionals. This broad target audience and strong terminological conformance with model theory distinguish MODA & MODE from classical software product line engineering approaches. Whilst the components of the methodology that are concerned with domain analysis can be applied without the help of any sophisticated software tools, the domain engineering components of the methodology are best performed with the help of a dedicated software tool. The MODA & MODE approaches have a track record in industrial practice that extends back to 1994, with roots in software product line engineering and conceptual modelling. The concepts and techniques of the approach have been refined, simplified and aligned with established mathematical theories, systems theory, and empirical research into human psychology, enabling new forms of inter-disciplinary collaboration between domain experts. This chapter provides an overview of MODA & MODE, and it also traces the most important concepts and techniques back to their origins.

Keywords Agents • Category theory • Collaboration • Commonality and variability analysis • Denotational semantics • Instantiation • Knowledge engineering • Metalanguage • Model validation • Modularity • Recursion • Value chains • Viewpoints

J. Bettin (✉)
The S23M Foundation, Melbourne, VIC, Australia
e-mail: jorn.bettin@s23m.org

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_11, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

Translating the tacit knowledge, which often resides within the heads of domain experts, or which is sometimes buried in difficult-to-maintain software code, into explicit knowledge that does not decay over time is a major challenge for most organizations. This chapter starts with a background section (Sect. 2) that provides an overview of disciplines, approaches, and theories that have influenced the conceptual foundation of the model oriented domain analysis and engineering (MODA & MODE) approach. The objective is to highlight the range of perspectives that are required when formalizing the knowledge and insights of domain experts that collaborate in the context of a complex value chain [21] that may stretch over multiple organizations. Section 3 of this chapter explores the way in which semantics emerge and incrementally evolve as a result of goal-directed human collaboration. Section 4 discusses the complexity resulting from the dynamic aspect of semantics in terms of interacting agents, perspectives, and the jargons that are used in the communication between agents. Section 5 describes the concepts and principles employed by the MODA & MODE methodology to achieve a modular and maintainable representation of formal knowledge. Section 6 outlines how MODA & MODE uses category theory to provide a robust framework for representing the structures encountered in any domain. Readers interested in further details are pointed to complementary literature and presentations. Section 7 covers the topic of domain analysis and describes specific techniques for eliciting and validating expert domain knowledge. Section 8 shifts the focus towards the bigger picture and the synthesis of the results of domain analysis in complementary or overlapping disciplines, building on the observations on jargons and denotational semantics made in Sect. 4. Given the increasing popularity of agile approaches in the management of software development and the success of lean production techniques in industrial manufacturing, Sect. 9 shows how MODA & MODE bridges the gap between agile approaches and lean methodologies. The latter methodologies and MODA & MODE share a common heritage in terms of W. Edward Deming's system of profound knowledge. Section 10 concludes with an outlook relating to future work on tool support for MODA & MODE.

2 Background

Whilst the techniques used in MODA to record the results of domain analysis have their roots in conceptual modelling, the process used to conduct variability and commonality analysis in MODA can be traced back to software product line engineering methodologies. The solution design and implementation techniques in MODE build on the formal models delivered by the MODA part of the approach, and apart from the emphasis on binding times, have little in common with the

program code centric approaches to variability management encountered in some software product line engineering methodologies [27].

Although the field of domain engineering has a long history, until recently applications in industrial practice were largely confined to the development of very large and expensive families of systems [10].

One challenge is to bridge the gap between theory and practice found in the education of computer scientists, mathematicians, and engineers. It is easy to invent hard problems for academic research, but it is hard to focus on those problems that actually reflect the needs of software intensive businesses. The Web, social networks, and mobile computing devices generate data in huge quantities, and lead to highly interconnected digital value chains. The problem of transforming the flood of raw data into actionable knowledge and valuable products is very different from the problem of automating previously manual business processes, which characterized the role of software in the last century. MODA & MODE are designed from the ground up for conceptual modelling of complex systems and make no distinction between models of "data" and models of "programs".

Another, and possibly even bigger, challenge originates from the prevailing approach to management and software development used by software intensive businesses. The MODA approach takes into account the observations of W. Edwards Deming on teamwork [12] and on the detrimental effects of classical management techniques in the context of complex systems with emergent behaviour. It is no longer useful or valid to think of a typical software system in isolation—every piece of software must be considered within the context of a wider value chain that in most cases extends beyond the boundary of a single organization.

The vast majority of artefacts that flow in a digital value chain are information artefacts produced and consumed by software users, and only very few digital artefacts that are exchanged between organizations have been produced by software development professionals. This means that all *software development* methodologies underrate the role of software users and stakeholders beyond the organizational boundary. Popular agile methodologies such as Scrum or Extreme Programming (XP) [16] emphasize the need for working together with customers on a daily basis but are mainly concerned about the quality and usability of the software programs that get developed, rather than the semantic quality and value of the data that users create, manipulate, and exchange. Complex networks of Web services result in unforeseen emergent behaviour and outages, often with significant financial impact on a large number of businesses and consumers.

Members of the S23M team have applied the MODA & MODE approaches in the following industries: telecommunications, electricity, industrial automation, mass customized industrial production, warehousing and logistics, news and media, government services, insurance, banking, and software. The domain experts who have been trained in or have been exposed to the methodology include: telecommunications engineers, electrical engineers, pricing analysts, supply chain management experts, Web content managers, actuaries, product managers, marketing experts, software product line architects, software product configuration experts.

2.1 Systems Theory

The Web that connects modern businesses does not have the neat tree structure of a simple value chain; it contains numerous feedback loops. The reality of intensive cross-disciplinary collaboration involves an increasing proliferation of open source software [22] and open data, and a blurring of organizational boundaries. Social networks and related technologies provide an excellent example of emergent system behaviour that spills over into all aspects of human life and economic activity. Furthermore each human individual is a complex system. The implication is that systems theory must be an integral part of any methodology for understanding and evolving modern organizational structures [13] as well as the software systems used for decision-making and collaboration. MODA & MODE view organizational structures as variabilities that are subject to a high rate of change and make use of feedback loops to validate the level of shared understanding between domain experts.

2.2 Deming's System of Profound Knowledge

W. Edwards Deming was an American management consultant who played an important role in the rebuilding of Japanese industry in the 1950s. He was ignored in the USA and Europe until the 1980s, when his work led to a proliferation of quality initiatives. Deming's theory and method [12, 13] is based on a system of profound knowledge that encompasses:

1. *Appreciation for a system*—A system consists of a set of interacting parts, a boundary, and the context that lies beyond the boundary. Consistent with systems thinking; Deming observes that a system generates emergent behaviour that cannot be predicted by analysing the behaviour of the parts, leading to a requirement to study the interactions of the parts in an experimental setting to gain deeper insight. The implication for conceptual modelling is that semantics are not inherent in a component but are generated incrementally, by new usage scenarios that involve the component.
2. *Knowledge about variation*—Understanding the causes of variation in the behaviour of a system is a prerequisite for making improvements. Deming's theory makes use of statistical observations to uncover the causes of variation, an approach that complements the model theoretic approach for variability analysis and validation that MODA applies in the realm of digital production.
3. *Theory of knowledge*—Deming emphasizes the critical role of domain-specific knowledge for improving a system. Software product line engineering professionals have come to the same conclusion, which is reflected in the pivotal role of domain-specific modelling languages in MODA & MODE for formalizing knowledge.

4. *Psychology*—Observations on intrinsic motivation lead Deming to conclusions about teamwork and productivity that in many cases clash with the teachings from mainstream business schools. MODA builds on Deming's observations by avoiding hierarchical representations of knowledge that obscure the observable flows of knowledge in organizations. MODA also considers empirical results on cognitive bias and collaboration [25, 26] from modern behavioural economics [15].

2.3 Software Product Line Engineering

The most important influence of software product line engineering on MODA is the concept of variability and commonality analysis [9, 27] and the strong and explicit distinction between domain engineering and application engineering [10]. MODA goes further in emphasizing the binding times of decisions and the need for synthesizing the different perspectives from complementary domain experts than most software product line engineering methodologies. Specifically MODA & MODE acknowledge that each individual domain expert has developed highly valuable mental models that are worthwhile being formalized in order to facilitate a shared understanding within the wider context of a value chain. MODA does not permit any formal model to exist outside the context of a concrete perspective or outside the context of a concrete binding time that relates to the role of a concrete domain expert.

2.4 Denotational Semantics

The use of MODA in many different contexts, from the development of computer aided software engineering (CASE) tools to organizational transformation in the context of mergers and acquisitions, has highlighted the critical importance of separating the activity of conceptual modelling from the activity of assigning names to concepts. Rigorous application of denotational semantics [24] was instrumental for the success of repository-based CASE tools, such as LANSA,[1] and continues to be a key differentiating factor between formal model-driven approaches to system analysis and design, and the use of models-as-a-sketch or the use of textual programming languages.

Emerging software tools that support MODA & MODE, such as the S23M platform [20] follow in the footsteps of the experimental Gmodel prototype [4] and completely de-couple the concern of modelling from the concern of identifying the concepts of a semantic domain as illustrated in Fig. 1. In a first step concepts are

[1] Lansa Inc., http://www.lansa.com.

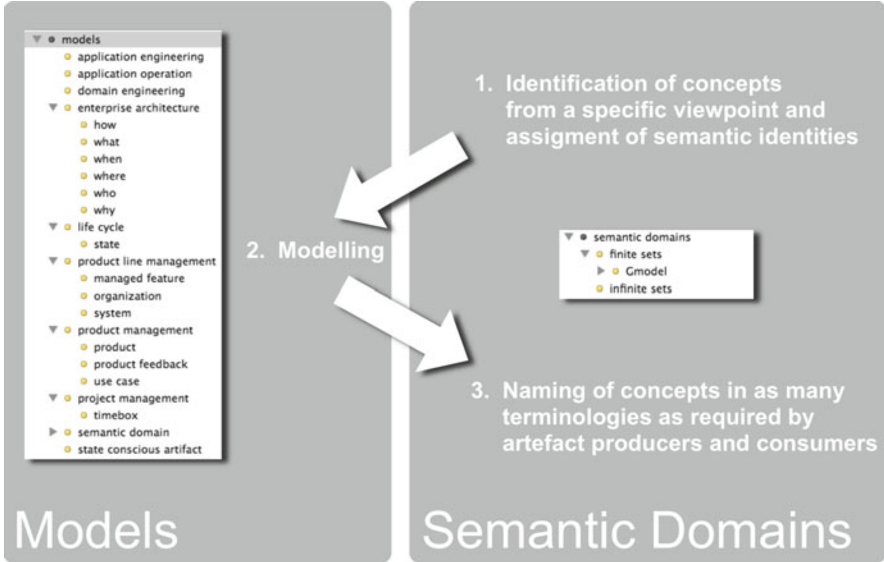


Fig. 1 Separation of modelling from naming in the S23M platform

identified from the perspective of a specific viewpoint and are associated with a semantic identity. In a second step concepts are referenced from multiple models to create formal structures. The assignment of names to semantic identities occurs in a third step, which is complexly independent of the model structures that have been created.

2.5 Model Theory

The inevitability of language evolution leads to the use of denotational semantics as well as to a need for powerful and simple tools for maintaining symbol sets and formal language definitions.

The mathematical definitions from model theory [14] provide a solid basis for the development of formal representations, which can range from the development of visual domain-specific modelling languages to the development of general purpose programming languages. MODA & MODE use model theoretic definitions and terminology to simplify software product line engineering terminology and the development of software tools that exploit mathematical theories such as category theory and Bayesian probability theory.

MODA & MODE acknowledge that domain-specific notations and terminology are subject to rapid evolution and take into account the social and commercial context in which domain-specific knowledge is maintained and shared. Experience

from the development of formal domain-specific languages in industrial settings [5,8] disproves the assumption that scientific research and established engineering jargon are the main source of terminology and symbol sets.

Terminological drift is the result of multiple unavoidable forces, ranging from the incorporation of new insights into a domain to changes introduced as a result of competitive marketing of products and services. Similarly, changes in domain-specific symbols and graphical notations can also be the result of accommodating new concepts or they can simply be an expression of artistic creativity of a product designer. The limited uptake of methodologies that advocate the use of formal models is at least partially due to naïve assumptions about the stability of notations and terminology.

2.6 Category Theory

Whilst model theory provides a comprehensive foundation for formal conceptual modelling, category theory ignores the complications introduced by the need for concrete notations and provides the substrate for abstract modelling [1], without any references to symbols or notations.

In MODA & MODE category theory is applied to formalize the concepts inherent in commonality and variability analysis. A powerful feature of commonality and variability analysis in MODA is the ability to introduce new parameters of variability from the viewpoint of a specific domain expert, without influencing the model structures used by other domain experts in any way. This is made possible by allowing every domain expert to create as many different perspectives and models as are needed to fully describe the specific domain, and by creating conceptual bridges between semantic domains rather than between models that pertain to different viewpoints.

2.7 The Biological Concept of a Cell

MODA practitioners have come to the conclusion that software engineering paradigms, programming languages, and current computer operating systems have very little to offer in terms of supporting modular construction of information artefacts. Apart from the digital codes that humans have developed, the only other possible source of inspiration for construction of modular information artefacts is found in biology, in the form of the biological cell and the genetic code.

> The cell is a true semiotic system, and the genetic code has been the first of a long series of organic codes that have shaped the whole history of life on our planet. [3]

Humans have only embarked down the road of significant dematerialization of artefacts in the last few years. In terms of producing scalable and resilient systems

biology is far ahead of any human attempts at modularizing the design of complex systems.

Software tools such as the S23M platform that implements the MODE approach use a module concept that is modelled on core characteristics of the biological cell, including the following features:

1. *Recursion*—The biological code is highly recursive, and the same is true of the encoding of the S23M Cell metalanguage.
2. *Behaviour*—The biological code uses recursion and proteins to encode cell behaviour, and similarly the S23M Cell metalanguage uses recursion and mathematical functions to encode the behaviour of a module.
3. *Modularity*—In biology the cell acts as a container for information and as a universal module concept. In particular cells are capable of self-replication. The Cell metalanguage also provides exactly one universal module concept, called the *cell*, which is applicable to all artefacts encoded in the Cell paradigm. In contrast many human designed modelling and programming languages provide multiple concepts that compete for performing the role of a module, such as the *class* and *package* concepts found in the unified modeling language (UML) and in a number of object-oriented programming languages.
4. *Remix*—Besides self-replication the biological code enables sexual reproduction, i.e., the combination of code from two separate individuals. The Cell metalanguage supports a concept of *instantiation* that emulates sexual reproduction. Instantiation leads to a new artefact that is defined in the context of the *structure* of a specific *category* artefact and the *perspective*[2] of a specific *container*[3] artefact.
5. *Blueprint*—In biology each cell contains the blueprint for the construction of a specific individual. In the Cell metalanguage, the blueprint for construction of an individual artefact is defined by the structure of the category of the artefact and is constrained by the perspective defined in the container of the artefact. References to category and container are an integral part of an artefact encoded in the Cell paradigm.

These characteristics are inherent in the structural pattern that is used in the implementation of the S23M Cell metalanguage and are available to users at all levels of abstraction and at all binding times.

2.8 Bayesian Probability

The entrenched distinction between humans and computers is becoming less and less useful. Humans are nondeterministic *learning* systems, but many software

[2] A *perspective* relates to the interpretation of an artefact by a user or a system.

[3] Every artefact (i.e., a MODE *cell*) has a *container* artefact.

systems are designed to be *deterministic*, i.e., non-learning. However, Artificial Intelligence software systems [23] are designed to be learning systems [19] just like humans.

Whilst humans will remain nondeterministic, we have a choice of designing highly deterministic or learning software components. Both kinds of components can be constructed from the same substrate, making use of the same conceptual foundation, for example the concepts encoded in the Cell metalanguage. The difference between deterministic and learning systems can be described in precise mathematical terms using graphs that contain feedback loops and Bayesian probabilities [11].

Deterministic systems can become quite complex, but always have the characteristic of a *tool*, whereas learning systems share characteristics with *biological entities*, including the ability to learn and evolve. Throughout history humans and a number of animal species have made use of tools, and tools continue to be extremely useful. Going forward an increasing number of nonhuman learning systems will join the ranks of tool users, and humans will have to learn to partner up with such systems.

The MODE approach to value chain and system design assumes that the best way to partner up with learning systems is to educate them and interact with them in ways that are most familiar to us: ways in which we interact with humans [25], determined by our sensorial capabilities and our cognitive limitations. In particular the approach to model validation described in Sect. 7.3 of this chapter can be used in conjunction with machine learning techniques and Bayesian probability.

3 Semantics

In order to improve the way in which humans collaborate and make decisions, there is no need for an empirically validated model of the human brain. Instead, it is sufficient to develop a mathematical model that allows the representation of concepts and meaning in a way that allows humans to share and compare parts of mental models.

MODE uses semantic domains as defined in denotational semantics for sharing symbol systems and associated meanings amongst humans, significantly improving the speed at which perceived meaning can be communicated, as well as the speed at which shared understanding [26] can be created and validated.

For most scientists this represents an unfamiliar use of mathematics, as meaning and understanding is not measured by an apparatus but is consciously decided by humans: The level of shared understanding between two individuals with respect to a specific model is quantified by the number of instances that conform to the model based on the agreement between both individuals. At a practical level the meaning of a concept can be defined as *the usage context of the concept from the specific viewpoint of an individual*. An individual's understanding of a concept can be

defined as *the set of use cases that the individual associates with the concept* (*consciously and subconsciously*).

These definitions are extremely helpful in practice. They explain why it is so hard to communicate meaning, they highlight the unavoidable influence of perception [15], and they encourage people to share use cases in the form of stories to increase the level of shared understanding. Most importantly, these definitions do not leave room for correct or incorrect meanings; they only leave room for different degrees of shared understanding—and encourage a mind-set of collaboration[4] rather than competition for "The truth".

3.1 Documenting Areas of Knowledge

In MODA domain knowledge is extracted by conducting workshops with a group of complementary domain experts and by observing the language and interaction patterns between the experts. In many cases simple tools such as whiteboards or flipcharts turn out to be an essential element for successful communication of knowledge between experts.

Whilst interactive workshops are commonly used for sharing insights and facilitating decision-making in software intensive organizations, it usually requires the presence of an experienced domain analyst to progress beyond conventional transfer of tacit knowledge between human experts to the stage where domain knowledge is formalized in a notation that is readily recognized and understood by domain experts, and that is at the same time suitable for machine-based processing.

The first step of formalizing domain knowledge consists of recording the concepts that are commonly used in expert discussions in relation to a particular problem domain. This step provides an ideal opportunity to confirm the extent to which a group of experts have already developed a shared terminology and to record any differences in terminology or implied semantics as far as this is possible at this early stage of analysis.

A frequent mistake made by even the most experienced software professionals is active contribution of new terminology that is not inherent in the language used by domain experts. Another frequently encountered mistake is the assumption that all domain experts should agree on a shared terminology across an entire product line or organization. This regularly leads to a simplistic official terminology that is plagued by ambiguous semantics, as, for example, a component in the context of software deployment is not the same as a component in the context of designing a user interface or in the context of designing an integrated circuit.

In MODA concepts are uniquely identified by a semantic identity, which can be either machine generated or assigned manually, and each concept can be associated with as many names as needed to correctly reflect the terminology used

[4] See http://www.slideshare.net/jornbettin/sharpening-your-collaborative-edge.

by different domain experts. Additionally, each concept can be associated with a domain-specific symbol, which is especially useful in scenarios where terminology is heavily overloaded, or where different experts use very different terminology. Joint development of shared domain-specific symbols is much more constructive than unrealistic attempts of standardizing terminology. Domain-specific symbols serve a key role in disambiguating terminologies and in creating conceptual bridges across disciplines.

3.2 Evolution of Semantics

It is important to realize that not only the terminology but also the semantics of a concept evolve over time. For example in an insurance company, each time new functionality that makes use of the "insurance policy" concept is developed, the semantics of "insurance policy" within the organization are extended. Additionally, the semantics of the concept "insurance policy" in company A differ from the semantics of the same concept in company B. Company A may only sell car insurance policies, and company B may have several lines of business. In the context of the systems of company A, there are no semantics for policies that relate to health insurance. Conversely, in the context of company B, the only semantics for car insurance policies are those that relate to the specific products offered by company B. These observations illustrate that semantic modelling in the context of a given organization differs from modelling in the Semantic Web,[5] which is an attempt to capture the common sense semantics that people associate with vocabularies in the public domain. The Semantic Web can be viewed as a lowest common denominator for semantic modelling in scenarios that involve interoperability between different organizations.

4 Modelling Value Chains in Terms of Interacting Agents

Conceptually software systems as well as human software users can be modelled as *agents* that interact in a value chain by producing and consuming digital artefacts.

The thin arrows in Fig. 2 represent semantic links embedded in digital artefacts. For clarity of representation typically different symbols are used to distinguish between human agents and software agents. Each agent in a value chain can play the role of artefact producer and artefact consumer in relation to other agents.

A *perspective* is characterized by the domain-specific *jargon* used to support communication between two agents. Jargon arises whenever the names assigned to the concepts used in the communication differ depending on the *viewpoint*. In

[5] The Semantic Web, www.w3.org/2001/sw/.

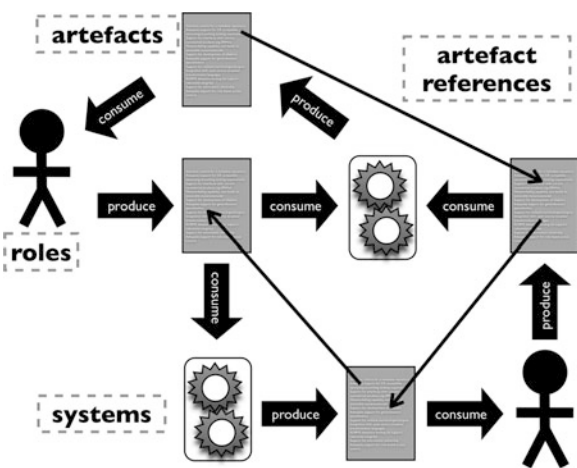


Fig. 2 Extract from a value chain involving agents A, B, X, and Z

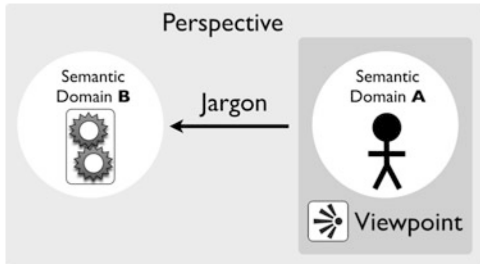


Fig. 3 Perspectives, viewpoints, and the origin of jargons

software tools that separate naming from modelling, one and the same artefact can easily be represented in as many different jargons as are required. In Fig. 3 the difference in the jargons between the two agents is represented by two distinct semantic domains.

In practice the scope of a domain-specific jargon is limited to the collaboration within a small team of peers that are working on a specific problem. Translations between jargons are required as soon as communication and collaboration with further teams or systems is required.

The concepts of agent, perspective, and jargon are explicit elements in all representations of knowledge in the S23M Cell metalanguage and are available for processing by software tools on demand.

Without reliance on denotational semantics, the communication challenges in a software intensive business that consists of hundreds of agents quickly leads to an explosion in complexity. The problem is familiar to anyone who has ever needed to integrate more than two software systems and to accommodate the terminological preferences of different groups of domain experts. When system integration is performed with conventional software tools that rely on textual specification and programming languages, the resulting code steadily but surely

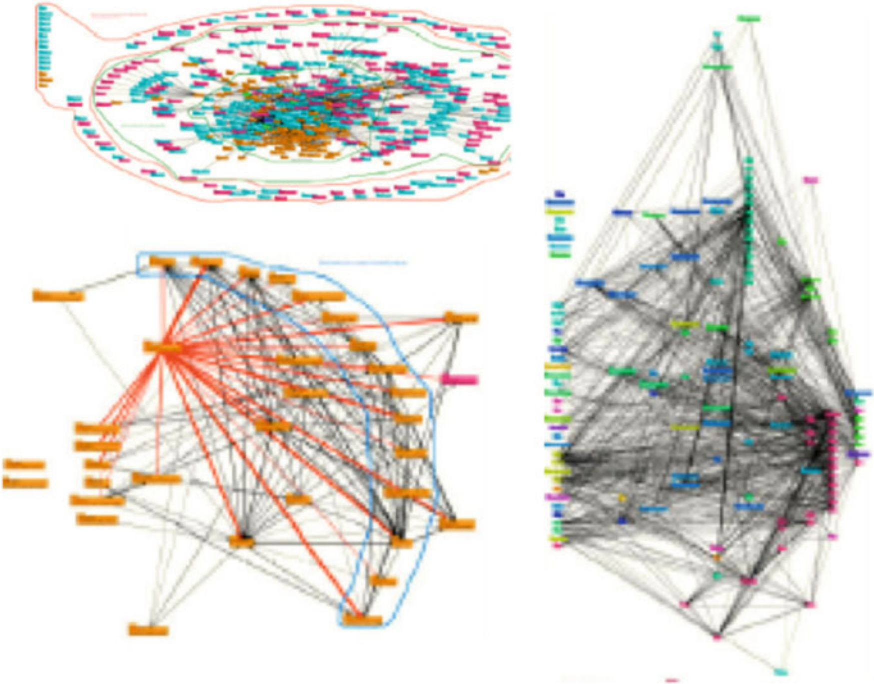


Fig. 4 Dependencies in a service-oriented architecture

takes on a form that is consistent with the Big Ball of Mud[6] architecture pattern. Figure 4 shows the result of a tool-assisted analysis of the dependencies between the state-of-the-art services and applications in the service-oriented architecture (SOA) operated by a large corporation. The pictures show heavy interdependencies between services, which defeat the goal of loose coupling, and which largely prevent independent deployment of individual services.

In contrast, systems that have been developed and integrated with CASE tools that manage multiple jargons with the help of denotational semantics often remain maintainable over periods of 10 years or more. Perhaps the strongest evidence for the effectiveness of denotational semantics comes from the pervasive use of automatically generated unique keys in relational databases and other storage technologies. Without the use of immutable semantic tokens, semantic links between data simply cannot be maintained in a reliable way. The content of large operational databases and data warehouses would be completely unmaintainable without the assistance of unique semantic tokens.

The S23M metalanguage enforces the workflow illustrated in Fig. 1, ensuring that all knowledge is accessible for processing in software tools in the

[6]Big Ball of Mud, http://www.laputan.org/mud/.

form of unambiguous semantic identities, and ensuring that additional symbolic representations of knowledge in a form that is intuitive for humans (e.g., text, icons, spoken languages, and instructional videos) can be added at any point in time, always taking into account the specific perspective and the language preferences of users.

5 The Role of Containers and Visibilities

MODA & MODE assume that every model is conceptually part of a container model, leading to a model containment tree structure. The top level of the containment tree consists of models that represent agents, i.e., individuals and systems. Beyond the top level, the containment tree is used as a structure to organize models (left side of Fig. 1) and to define the visibility of models in relation to other models (Fig. 5), with agent representations acting as the root for all models produced by an agent.

In order to extend the use of formal information models beyond the small community of domain engineering professionals, it is useful to tap into the vocabulary that is commonly used to describe workflows, business processes, and deliverables. Hence, instead of continuously talking about formal mathematical models, it is sensible to use the term artefact instead of model in the role of a container. The notion of an *artefact* can be explained to non-mathematically inclined domain experts as follows:

1. An artefact is a *container of information.*
2. An artefact is *created by a specific agent* (*a human or a system*).
3. An artefact is *consumed by at least one agent* (*a human or a system*).
4. An artefact represents a *natural unit of work* (*for the creating and consuming agents*).
5. An artefact *may contain links to other artefacts.*
6. An artefact *has a state and a lifecycle.*

When implementing tools for manipulation and management of formal models, the following technical considerations need to be added to the notion of artefact:

7. A formal artefact is created with the help of a software tool that enforces specific instantiation semantics, and it includes all the metadata and instantiation semantics that were used to produce the artefact (quality constraints).
8. The information contained in a formal artefact can be easily processed by software tools (in particular by transformation languages).
9. Referential integrity between formal artefacts is preserved at all times with the help of a software tool (otherwise the necessary level of completeness and consistency is adequate neither for automated processing nor for making business decisions based on artefact content).

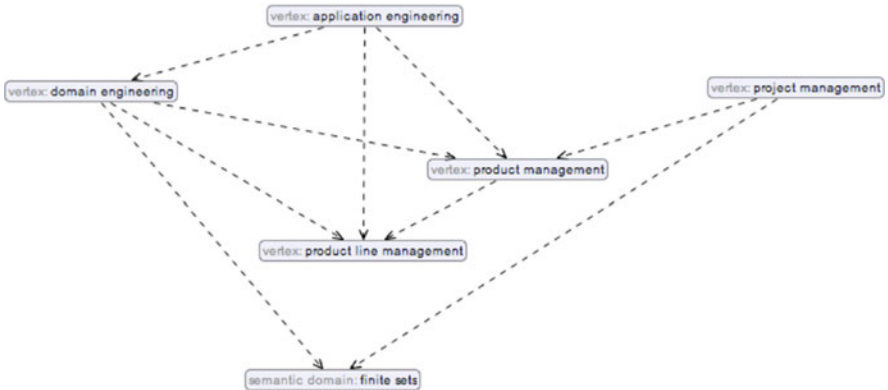


Fig. 5 Artefact visibility declarations

10. No circular links between formal artefacts are allowed at any time (a prerequisite for true modularity and maintainability of artefacts).
11. The lifecycle of a formal artefact is described in a state machine (allowing artefact completeness and quality assurance steps to be incorporated into the artefact definition).
12. The events consumed and produced by the artefact state machine are available for processing in software tools (allowing transformations to be triggered at all necessary points in time to keep artefacts synchronized with all derived artefacts).

It is important to note that the existence of a containment tree structure does not imply a universally applicable rigid hierarchical representation of models. Instead, the containment tree structure represents the organizational preferences of an individual agent. If agent A references models produced by agent B, the containment tree structure chosen by agent B has no impact on the organization of the models produced by agent A.

Using agents as root containers for semantic domains and models has significant practical benefits in terms of the stability of models over time. The average life expectancy of an association between an agent and an organization is much less than 10 years, whereas the life expectancy of human careers and of software systems is measured in decades.

Traditionally it is common practice to organize artefacts according to the legal organizations that own them, but this can create massive problems when organizations or business units are merged or divested. It often becomes practically impossible to undo the encoding of obsolete organizational structures in software source code and in database schemas.

It is much more appropriate to represent legal artefact ownership as a variable that can easily be changed and to consistently apply this approach to all information artefacts.

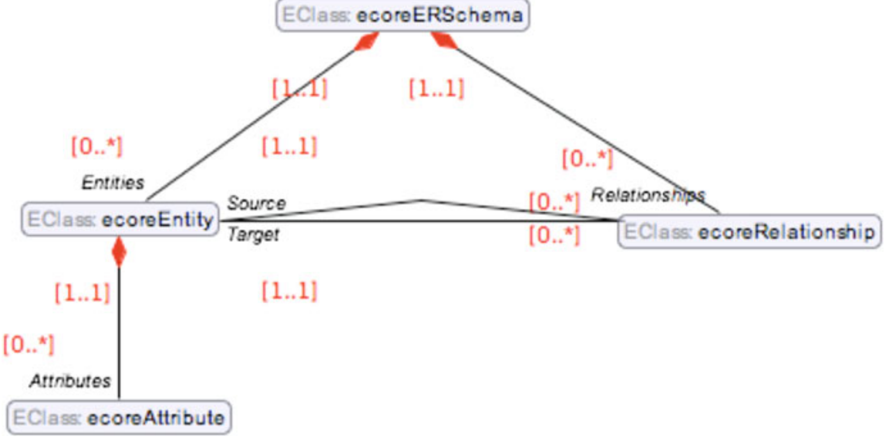


Fig. 6 Graph structure of a model decorated with instantiation semantics

In MODE visibilities are used as the mechanism to manage the allowable dependencies between models. Visibilities give designers of modelling languages an unprecedented amount of control over the models that language users can instantiate. In the experience of the author, such functionality is essential for managing the dependencies in large-scale software intensive systems.

6 Categories

Every digital artefact is a model of a theory in the model theoretic sense. The graph structure of the model is best represented in the form of visual diagrams. The diamond symbols at line ends and the expressions in square brackets in Fig. 6 are representations of the instantiation semantics that pertain to the model, and the names shown in grey are categories.

The model pattern known as the power type pattern in object orientation occurs pervasively in highly configurable systems. Closer examination of the power type pattern exposes it as a technical kludge[7] that forces the fragmentation of semantic identities and demonstrates the limits of the object-oriented paradigm, which is currently still treated as dogma by many software engineers.

By allowing multi-level instantiation as shown in Fig. 7, MODA & MODE eliminate the need for the power type pattern and avoid the fragmentation of

[7]Further details on the specific problems caused by the power type pattern are illustrated at www.slideshare.net/jornbettin/from-muddling-to-modelling.

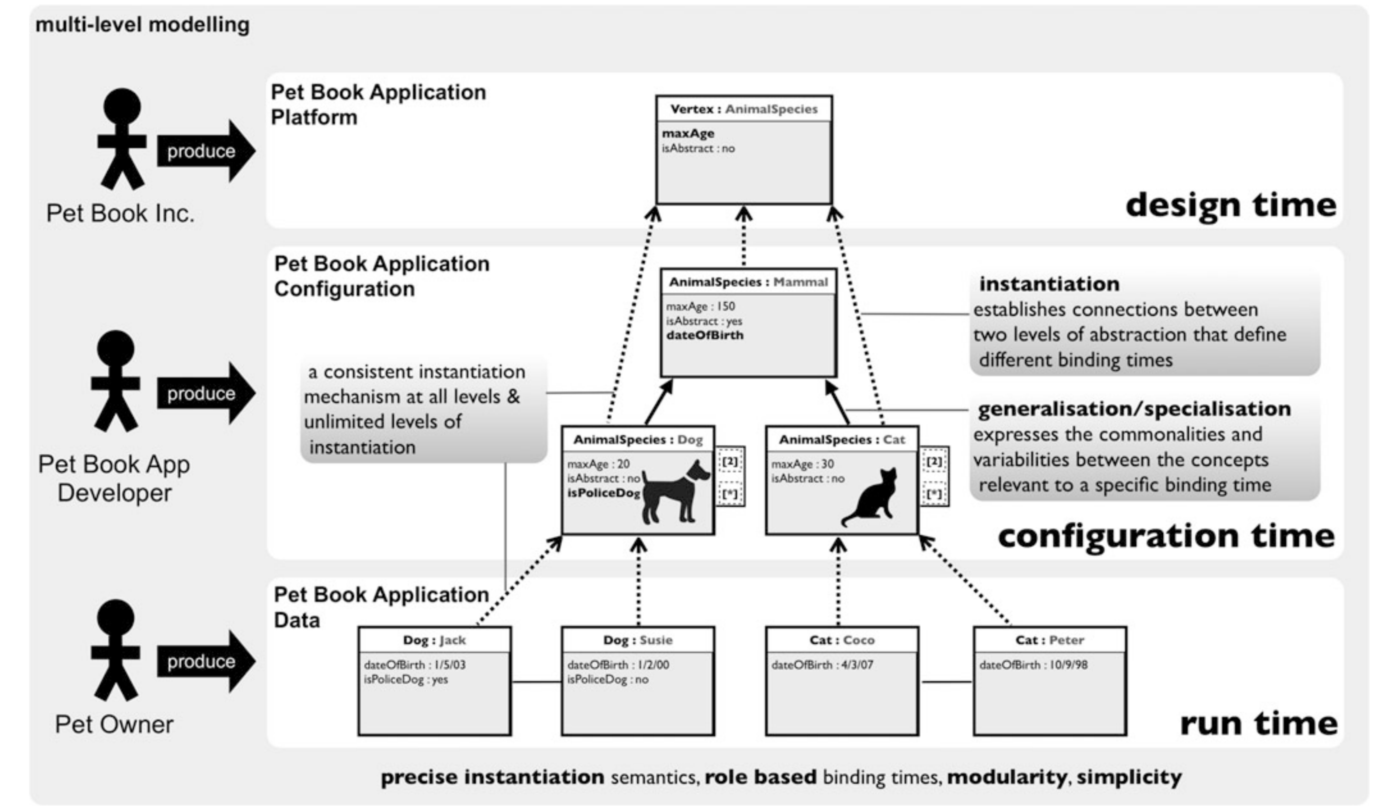


Fig. 7 Multi-level instantiation semantics

semantic identities. The S23M Cell metalanguage provides a uniform instantiation function that is available at all level of abstractions (binding times) to all users.

On the one hand, this leads to substantial simplifications in tool-based model processing, as the same tools that are used at design time can be used at configuration time, at run time, and at any other (fine grained) binding times that may apply in a particular context. On the other hand, the ability to decorate any artefact with instantiation semantics—an ability that is significantly required—reduces the need for model changes or updates. For example, decoration with instantiation semantics allows a model of a concrete product to evolve into a template structure for an entire product line.

Usually a new technology product initially tends to provide a simple set of functions that meet the demands of the first few customers. Over time, as the adoption of the technology product increases, as feedback is received from users, and as competitors start producing comparable offerings, the pressure grows to understand the specific use cases of individual customers, and to differentiate the product. Rather than polluting the formal specification of the original product with new abstractions and new binding times, it is often preferable to use multi-level instantiation to highlight the elements of the original model that need to become variabilities in the future product line.

As can be seen in the example in Fig. 7, the availability of multi-level instantiation does not eliminate the usefulness of the generalization/specialization construct that is offered in the UML and in object-oriented programming languages. In the MODE approach, the use of generalization/specialization is reserved for use within a given binding time, allowing a domain expert to articulate commonalities at a particular level of abstraction, without impacting any of the formal artefacts that relate to earlier or later binding times. Specifically the MODE approach limits the scope of object-oriented inheritance of functionality to the elements associated with a particular binding time. Any tools that process a model artefact expressed in the S23M Cell metalanguage can easily access functionality at different binding times by navigating up or down the multi-level category tree.

Multi-level instantiation [2, 4, 7, 17, 18] is a very effective tool for representing the results of variability and commonality analysis, as it simplifies the representation of complex products, and leads to visual diagrams that are intuitive for humans to understand.

7 Variability and Commonality Analysis

The success of domain engineering critically depends on two factors: firstly on the availability and access to domain expertise, and secondly on the quality of variability and commonality analysis. Whilst the first factor is very hard to influence in the short term, the second factor is a function of the experience of the domain analyst and the quality of the techniques used in domain analysis to uncover and validate domain knowledge.

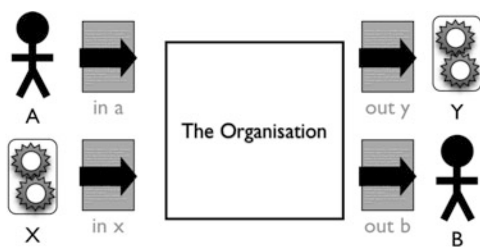


Fig. 8 Value chain context diagram

One of the best introductions to the reality of variability and commonality analysis is found in [9]. Often the need for domain analysis is triggered by a commercial imperative, such as the need to launch a specific product by a fixed target date. The product in question can be an insurance product, an industrial automation system, an automobile, or a commercial software product—in all cases the domain analysis challenge is the same: finding the simplest possible representation for all the different aspects of the product specification and subjecting the specification to thorough validation by all relevant domain experts.

The term *variability and commonality* analysis is not well established beyond the domain engineering community, and often it is more appropriate to talk about *value chain analysis*, especially if senior business executives need to fund the activity and perhaps even participate in the activity.

Ideally domain analysis does not lead to any need for participants to learn new terminology, and ideally domain analysis techniques are tailored for unobtrusive insertion into product or service design processes. MODA & MODE achieve this goal by exploiting denotational semantics and by offering support for individual viewpoints and as many jargons as required.

7.1 Value Chain Analysis

Value chain analysis should always start from the outside in, and the initial objective is not to capture how a business operates but to identify what it delivers and what external inputs are required in the process. Limiting the initial focus on deliverables and input artefacts that cross the organizational boundary leads to a very concise description of the purpose of the organization. A good way to record this description is a diagram that shows the organization as a box that is surrounded by external supplier and customer roles (agents), and arrows between the box and the various agents that are annotated with the names of the artefacts that are being exchanged. Such a diagram (Fig. 8) is often called a context diagram. It is immaterial what notation or tool is used, what matters is the reliability of the information and the use of unambiguous names for agents and artefacts.

If the number of artefacts gets too large, then perhaps some artefacts can be grouped together to a set, and the set can be assigned an appropriate name, or

perhaps some artefacts are simply variations of a common theme, and the context diagram only needs to show the generalization. Similar rules apply if the number of agents gets too large.

Once the context diagram has been established, the next steps of the analysis follow the following pattern:

1. *Identifying those artefact exchanges that warrant a closer investigation in terms of the objective of the analysis*, such as the need to deliver a new product or the need to increase the speed of service delivery. If the biggest pain point is not obvious, each artefact exchange can be annotated with further information that illustrates the value and cost of the interaction (performance indicators such as the frequency with which artefacts are exchanged, the cost of producing an artefact, the time to produce an artefact, the percentage of artefacts that do not meet quality expectations, etc.). Another issue that can surface at this stage in the analysis is the realization that different business units use different names for artefacts and agents, and that parts of the organization lack a clear vocabulary. This is the opportunity for the domain analyst to immediately address the issue and to facilitate the discussion between domain experts.
2. *Recording the extract from the context diagram that contains the artefact exchanges that need to be analysed in more detail*. The resulting picture is extremely important to provide clear objectives and guidance to those tasked with further analysis. Otherwise the analysis may result in costly business process wallpaper that offers no new insights.
3. *Analysing the internal processes used to consume the input artefacts or to produce the output artefacts that warrant further investigation*. In the analysis, the same guidelines are applied as for the initial context diagram—keeping the picture simple, ideally to a single page (Fig. 2). Again it is a matter of carefully selecting the most appropriate level of abstraction for the representation. The result typically does not need to contain detailed workflow information such as conditions and business rules. Instead the diagram should be annotated as needed with the same kind of high-level meta information that was used to decide on the focus of the analysis. Once the rationale for condensing workflow information into a small set of performance indicators is understood, it is obvious how to proceed with further analysis—by selectively drilling into areas that require attention.

7.2 The Six Domain Analysis Questions

Variability and commonality analysis in MODA is guided by six questions that are designed to elicit domain knowledge from the participating domain experts.

1. How often does the decision require revision?
2. Who makes the decision?
3. When is the decision made?

4. Where, in which artefact is the decision made?
5. What are the possible choices for the decision?
6. What heuristics apply when implementing the decision?

The six MODA questions and the supporting explanations in Fig. 9 are designed specifically with the intent of uncovering the information needed to translate the set of variabilities associated with a domain or a series of connected domains into a modular set of formal abstract syntax specifications for domain-specific modelling languages.

The effectiveness of the MODA approach to modelling language design has been shown in numerous organizations and in a range of industries.[8] Typical results include a reduction in specification artefact size by factors between 3 and 20, largely due to significantly improved modularity and the use of better abstractions. Uncovering the latter typically requires very little initiative by the domain analyst. Often the "new" abstractions can be taken straight from the vocabulary of domain experts—which says a lot about the quality of earlier system designs that have been developed with traditional *software development* methodologies.

One of the main roles of the domain analyst in the formalization of knowledge lies in the confirmation of domain boundaries and binding times. As observed by W. Edwards Deming, improvements require a combination of domain knowledge and external expertise in applied systems theory and complexity management.

7.3 Validation via Instantiation

When designing the formal structure of artefacts, it is essential that both the artefact producers and a sufficient number of artefact consumers will be involved in validating how well the models that have been developed meet the needs of all the agents that work with the models.

In MODA & MODE a significant amount of validation can easily be performed immediately after a model has been created, as MODE tools, such as the S23M platform, enable users to instantiate any model and to visualize the instances created in either a graphical or a tabular representation.

As already pointed out in Sect. 2.1, words such as "component" are often associated with very different semantics by different domain experts, even within a single organization. Validation via instantiation offers a very effective mechanism to uncover misunderstandings and clarify whether a "car" refers to a specific product line, a specific version of a product line, a partially specified car model within a product line, or a representation of a concrete instance of a car with a specific serial number.

---

[8] See case study summaries on pages accessible from http://www.s23m.com.

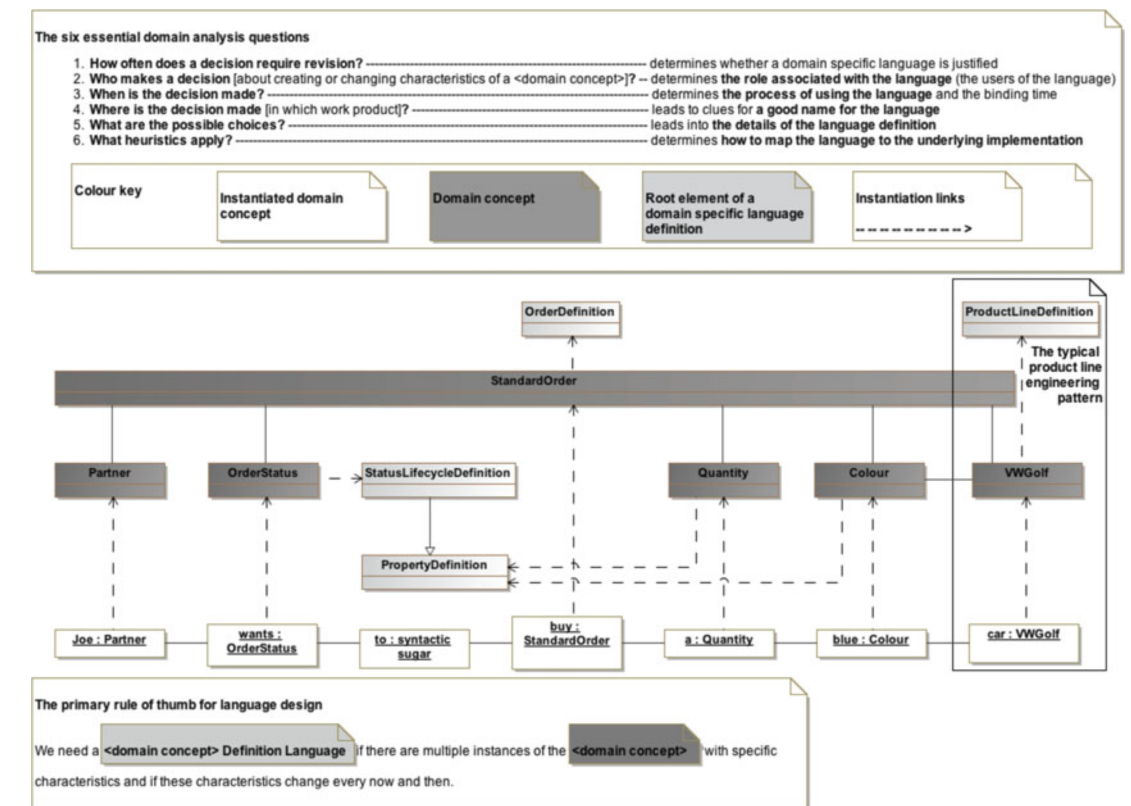


Fig. 9 The six domain analysis questions in the context of a car manufacturer

By requiring domain experts to identity several example instances of each concept in their domain, any mismatches in assumptions about the level of abstraction are quickly resolved. The effectiveness of validation is further improved when the software tool that is used is not only able to record the instances but also able to connect instances of various categories into a cohesive model that domain experts can relate to.

8 Domain Synthesis

Beyond model validation via instantiation MODA & MODE make use of a number of further techniques to improve model quality, to facilitate cross-disciplinary collaboration between domain experts, and to increase the degree of automation across an end-to-end value chain.

8.1 System Integration and Reuse of Concepts

The separation of modelling from the identification and naming of concepts enables reuse of concepts and semantic domains independently from reuse of models. This kind of functionality has been available for a very long time in CASE tools such as LANSA but is still missing from many modelling tools and programming languages.

In addition to the reuse of semantic domains, the S23M platform provides explicit support for different equivalence transformations that can be applied when using tools to perform model-based computations. In particular the Cell metalanguage allows users to declare concepts that have been defined independently in different domains to be semantically equivalent, or if so desired, to define specific equivalence classes of concepts that are only applicable from their specific viewpoint. These techniques open new approaches for system integration and software interoperability [6].

8.2 Collaborative Symbol Development

In all scenarios where domain vocabularies contain words that are heavily overloaded, or simply in all those scenarios where the objective is to represent domain knowledge in a graphical user interface, it is necessary to make use of appropriate visual symbols. Although widely used consumer software such as operating systems generally make use of high quality symbols, it is not uncommon for enterprise software to rely on poorly designed or overly generic symbols. The most problematic cases are encountered in the context of custom business software systems developed for use within a single organization.

Based on these observations and based on experience in designing visual domain-specific modelling languages in collaboration with domain experts, the author has come to the conclusion that collaborative symbol development is a highly beneficial technique that should be encouraged in all organizations that develop software. The positive effects of collaborative symbol development include:

1. Domain experts are prompted to think about the core characteristics of a concept, facilitating discussion about use cases that involve the concept and raising the level of shared understanding between domain experts.
2. Involving domain experts in symbol set design immediately increases the level of intuitiveness of all software solutions that make use of these symbol sets, which leads to a whole range of positive downstream effects on software quality and data quality.
3. Discussion about visual symbols can highlight characteristics and structures that are associated with specific semantics, and which are hence worthwhile capturing in a formal model.
4. Graphical designers can apply their creativity constructively, without needing to make speculative assumptions about the essence of a concept.
5. All those cases where a concept represents a concrete item in the physical world are quickly identified and confirmed, and symbol development can be short-circuited by sourcing a simple photographic image.

8.3 Visual Modelling Language Design

Once the abstract syntax of the modelling languages required in the context of a particular domain and organization has been confirmed, and once all the underlying semantic domains include appropriate symbol sets, all the basic ingredients for visual domain-specific modelling languages are in place.

Further steps to define concrete syntax are limited to the default arrangement of symbols in relation to one another, and to the notation used to represent the edges that connect the domain specific symbols. The finer details of concrete syntax design are often best addressed via one or more small domain-specific languages for concrete syntax design. The details of these concrete syntax design languages tend to depend on the fundamental patterns that govern the user interface technology that is being targeted.

9 The Connection to Agile and Lean Approaches

This chapter provides an overview of key concepts and techniques in MODA & MODE, but it may leave readers with a background in computer science and software engineering wondering how the approach relates to popular software development methodologies.

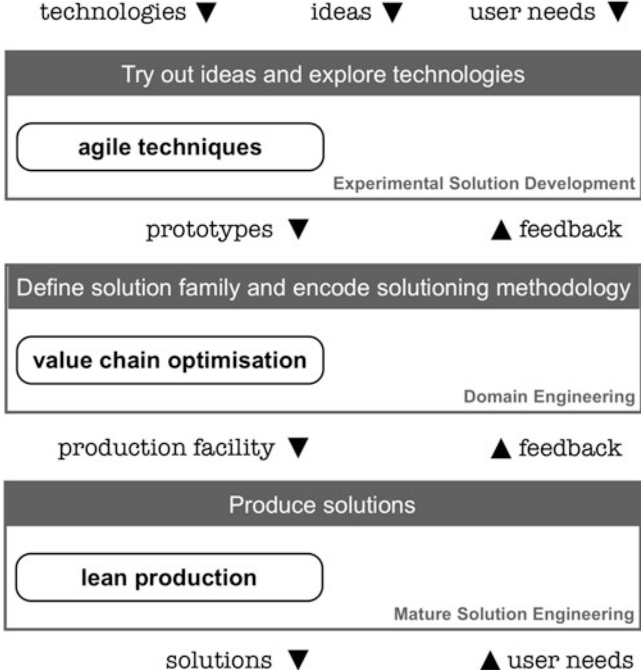


Fig. 10 Domain engineering in relation to agile approaches and lean production

Domain analysis and engineering provide a framework for exploiting deep domain knowledge to achieve substantial gains in productivity and quality within an end-to-end value chain. Whereas classical Enterprise Architecture frameworks such as Zachman or TOGAF offer a standardized set of tools for architectural design to address needs across all business support processes, MODA offers analysis techniques for streamlining the core of a business, and MODE offers engineering techniques for creating automation tools. Figure 10 shows how domain engineering addresses the conceptual gap between agile techniques and lean production techniques.

Agile techniques and experimental solution development leads to prototypes that can be refined and optimized by domain engineering. The resulting highly automated production facility provides the basis for a lean approach to production.

The reason for the success of agile techniques lies in the full recognition of the risks inherent in the development of solutions that involve unfamiliar implementation technologies or unfamiliar domains. Techniques such as time-boxing, short iterations, and the validation of working software following each iteration minimize the impact of misunderstandings between stakeholders and the project team. They further provide excellent opportunities for gathering initial knowledge about a new technology or an unfamiliar domain.

Lean production techniques complement agile techniques by focusing on the learning opportunities provided in a production environment that is characterized by repeatable processes in a familiar domain—specifically by the availability of performance metrics from processes that are under statistical control.

The MODA & MODE approach allows the deep domain knowledge that is gained in a lean production environment to be formalized and to be used in the context of further process automation and optimization.

As needed, especially when unfamiliar implementation technologies are introduced into the picture, the MODE approach makes use of agile techniques to develop and validate experimental prototypes before proceeding with the formalization of heuristics and with advanced process automation.

The advantage gained by formalizing domain knowledge with MODA & MODE lies in the quality attributes of the representations produced by the approach, in particular in terms of modularity, a concept terminology that not distorted by underlying implementation technologies, and in terms of improved maintainability. Empirical results show that improved modularity and better abstractions typically reduce the size of specification artefacts by a factor between 3 and 20. These factors have been observed when comparing extant configuration artefacts or program code with corresponding formal models that were developed with the MODA & MODE approach.

As pointed out in Sect. 1, the primary target audience for MODA & MODE consists of domain experts in any knowledge-intensive discipline, which has a significant influence on the tools that are being developed. The vast majority of software users, who are producing large amounts of informal and formal specification data, have an increasing need to attach precise semantics to the artefacts they are producing and consuming.

10 Conclusions and Future Work

The MODA & MODE methodology brings together elements from systems theory, the foundations of mathematics, human psychology, and empirical knowledge from software product line engineering. Industrial applications in several industries demonstrate that these elements fit together extremely well and are capable of delivering results that compare very favourably with the results achieved via other methodologies, many of which require the use of a patchwork of software tools and often do not make use of a formal mathematical foundation.

The methodology and related S23M tools are currently being extended to support Bayesian probability models and the specific concerns of learning systems. This work largely involves the creation of appropriate formal artefacts in the Cell metalanguage rather than coding in traditional programming languages.

Significant further work lies ahead in refining the software tools, in particular the development of a high quality Web-based graphical user interface. This work involves binding traditional user interface technologies and programming languages

to the abstract representations exposed by the S23M Cell Application Program Interface, as well as a significant amount of graphical design.

References

1. Adamek, J., Herrlich, H., Strecker, G.E.: Abstract and Concrete Categories – The Joy of Cats. Dover Publications, New York (2004)
2. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. IEEE Trans. Software Eng. **35**(6), 742–755 (2009)
3. Barbieri, M.: "Life is semiosis – the biosemiotic view of nature" cosmos and history. J. Nat. Soc. Philos. **4**(1), 29–52 (2008)
4. Jorn Bettin, Tony Clark: Advanced modelling made simple with the Gmodel metalanguage. In: Proceedings of the First International Workshop on Model-Driven Interoperability, pp. 79–88. ACM (2010)
5. Jorn Bettin, Tony Clark: Gmodel, a language for modular meta modelling. In: Australian Software Engineering Conference, KISS Workshop, Gold Coast, 14–17 April 2009
6. Jorn Bettin, William Cook, Tony Clark, Steven Kelly: Knowledge industry survival strategy (kiss): fundamental principles and interoperability requirements for domain specific modeling languages. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, pp. 709–710 (2009)
7. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling: A Foundation for Language Driven Development. Ceteva, Sheffield (2008)
8. Clark, T., Sammut, P., Willans, J.: Superlanguages: Developing Languages and Applications with XMF. Ceteva, Sheffield (2008)
9. Cleaveland, C.: Program Generators with XML and Java. Prentice-Hall, Upper Saddle River, NJ (2001)
10. Clemens, P., Northrop, L.: Software Product Lines – Patterns and Practices. Addison-Wesley, Boston, MA (2002)
11. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press, New York (2009)
12. Edwards Deming, W.: Out of the Crisis. Massachusetts Institute of Technology, Cambridge, MA (1982)
13. Edwards Deming, W.: The New Economics for Industry, Government, Education, 2nd edn. The MIT Press, Cambridge, MA (2000)
14. Hodges, W.: A Shorter Model Theory. Cambridge University Press, New York (1997)
15. Kahneman, D.: Thinking, Fast and Slow. Farrar, Straus and Giroux, New York (2011)
16. Kniberg, H.: Scrum and XP from the Trenches. C4Media, Toronto (2007)
17. Laarman, A.: An Ontology-Based Metalanguage with Explicit Instantiation. University of Twente, Enschede, The Netherlands (2009)
18. Alfons Laarman, Ivan Kurtev: Ontological metamodeling with explicit instantiation. In: van den Brand M., Gasevi D., Gray J., (eds.) Software Language Engineering. Lecture Notes in Computer Science, vol. 5969, pp. 174–183 (2010).
19. Mitchell, T.M.: Machine Learning. McGraw-Hill, Maidenhead (1997)
20. Open source S23M platform code at https://github.com/s23m (2012). The S23M Foundation
21. Porter, M.: What is strategy? Harv Bus Rev **November–December**, 61–78 (1996)
22. Rosen, L.: Open Source Licensing – Software Freedom and Intellectual Property Law. Prentice Hall, Upper Saddle River, NJ (2005)
23. Russel, S.J., Norvig, P.: Artificial Intellingence, A Modern Approach. Prentice Hall, Upper Saddle River, NJ (1995)

24. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. William C. Brown Publishers, St. Louis (1986)
25. Tomasello, M.: Why We Cooperate. The MIT Press, Cambridge, MA (2008)
26. Michael Tomasello, M., Carpenter, J.C., Behne, T., Moll, H.: Understanding and sharing intentions: the origins of cultural cognition. Behav Brain Sci **28**, 675–691 (2005). http://email.eva.mpg.de/~tomas/pdf/BBS_Final.pdf. Accessed 2010
27. Weiss, D., Lai, C.T.R.: Software Product Line Engineering: A Family-Based Software Development Process. Addison-Wesley Professional, Reading, MA (1999)

Multi-Level Meta-Modelling to Underpin the Abstract and Concrete Syntax for Domain-Specific Modelling Languages

Brian Henderson-Sellers and Cesar Gonzalez-Perez

Abstract A domain-specific modelling language can be considered as a situationally focussed conceptual modelling language. A modelling language is typically underpinned by a meta-model that defines its abstract syntax, utilizes a notation (a.k.a. concrete syntax) and possesses a well-defined semantics, sometimes with an associated ontology. However, the relationships between models, meta-models, modelling languages and ontologies are not well defined in the literature. In particular, the implications of the strict meta-modelling paradigm fostered by the OMG in relation to the type/instance duality are often described in a vague and equivocal fashion. This chapter provides a solid theoretical foundation for the construction of domain-specific modelling languages that can help define both the abstract and concrete syntax aspects. Two example languages are described: ISO/IEC 24744 (Software Engineering Meta-model for Development Methodologies), a language that can be used to define software-intensive development methods; and FAML (FAME Agent-oriented Modelling Language), a language for the specification of agent-oriented software systems.

Keywords Concrete syntax • DSL • Meta-models • Modelling • Ontologies

---

B. Henderson-Sellers (✉)
School of Software, University of Technology, Sydney, PO Box 123, Broadway, NSW 2007, Australia
e-mail: Brian.Henderson-Sellers@uts.edu.au

C. Gonzalez-Perez
Institute of Heritage Sciences (Incipit), Spanish National Research Council (CSIC), San Roque 2, 15704 Santiago de Compostela, Spain

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_12, © Springer-Verlag Berlin Heidelberg 2013

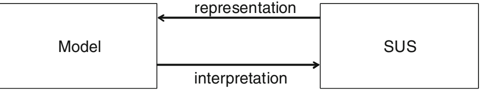


Fig. 1 Relationships between a model and the SuS (after [39])

1 Introduction

Domain engineering focusses on reuse within a specific business domain of interest and is often stated to encompass topics such as reuse, e.g. [23, 36], product line architectures and domain-specific languages, e.g. [10]. Domain-specific modelling languages (DSMLs) allow users to work with a set of concepts closely related to their domain-specific knowledge, concepts that may be encapsulated as a domain ontology [23, 57]. In other words, DSMLs are used to describe and document models that are constructed as part of software development for applications within a highly specialized business or application domain, e.g. [25].

Modelling languages in general must be formally defined and these definitions are often said to comprise the modelling language (ML) itself. This ML is often expressed using a meta-model. In studying domain engineering, there is thus a need to comprehend these elements: models, meta-models, ontologies, and modelling languages—and how they all fit together.

Modelling is all about abstraction. It is often said that a model is a representation of a (part of) reality, called here the SuS or system under study (Fig. 1). That reality could be at one of a range of granularities including a specific software system or a specific domain of interest (Fig. 2). In a software development context, the SuS is often described using a general purpose software engineering modelling language—a widely used example is the Unified Modeling Language, UML [70]. When the SuS is a highly specific domain, such as banking or healthcare, the language that is used to describe entities in this specialized domain is known as a DSML.

To understand DSMLs, we need to understand how they are related to more general modelling languages (MLs) and also to the broader context of domain ontologies. Figure 3 shows that, in the former case, we can simply state that a DSML is a specialized type of ML—although it should not be overly specialized—nor overly general [51]. A domain ontology is a description of the entities in a specific domain and is therefore a kind of model—in this case one that is described not by a general-purpose modelling language but by a DSML. Both DSMLs and general purpose MLs are used to describe a model—or, conversely, that a model uses a modelling language (Fig. 4) for its representation and, hence, its communication to others. These models may refer to endeavours (e.g., specific projects), to the method domain (of methodology and tools) or to the meta-model domain (Fig. 5). In each case, the description of the model uses the modelling language specification, which defines the ML—either in terms of a meta-model, BNF, a grammar, etc., e.g. [52]. Thus, to understand DSMLs in detail, we also need to discuss meta-modelling.

This chapter investigates the links between models, meta-models, DSMLs and ontologies in the context of their mathematical underpinnings (cf. [39]). In Sect. 2,

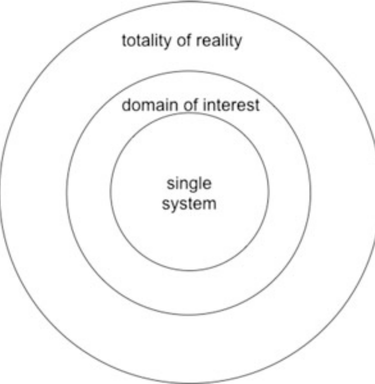


Fig. 2 Three areas of interest: a single system, a larger domain of interest (a specific technical domain, for instance) and reality. Any of these could be called the system under study (SUS) (after [39])

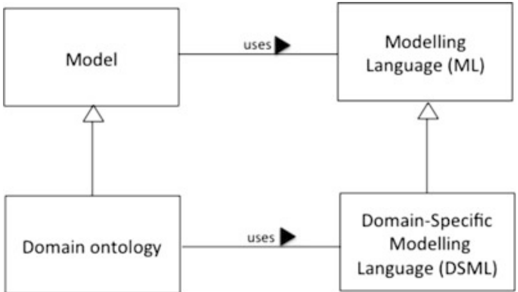


Fig. 3 The role of domain-specific modelling languages (modified from [39])

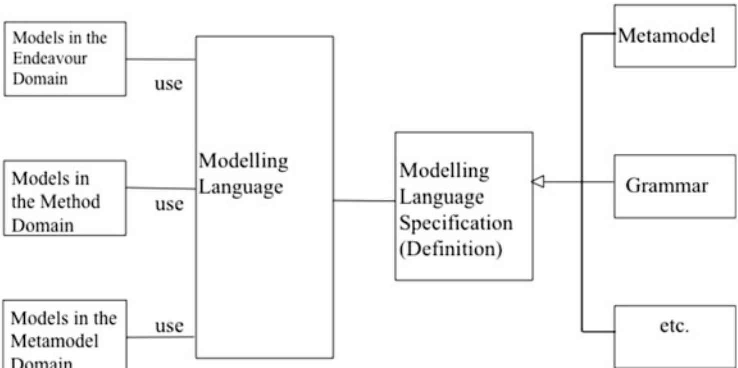


Fig. 4 Three domains defined by the category of users. For models created in any one of these domains, a modelling language (ML) is used. That ML is often the same for the three layers and can be defined in one of a number of ways

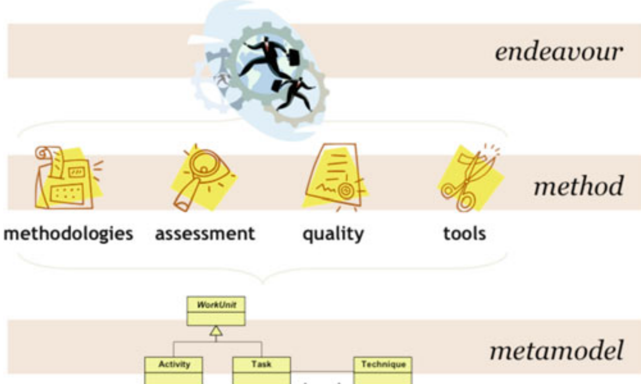


Fig. 5 Three level metamodel architecture as used in [45]. This architecture is based on practice rather than theory (cf. Fig. 6). People work on real projects using methodologies, tools etc., all of which are defined in the so-called "meta-model domain" (after [38])

we present the elements of the "traditional" architecture for meta-modelling followed by basic relevant mathematical structures in Sect. 3. Section 4 discusses models and meta-models before utilizing these for DSMLs in Sect. 5. Section 6 introduces two DSML case studies: for software development methodologies and for agent-oriented modelling.

2 Traditional Architecture

In software engineering and conceptual modelling, the traditional (at least over the last two decades) architecture has been the OMG's four layer strict meta-modelling construction (Fig. 6). This argues for the four so-called abstraction levels connected by instanceOf relationships. This choice of type-instance connections between pairs of layers is known as strict meta-modelling [4, 5]—an approach that forbids use of instanceOf within any single layer—seen by many authors as problematical, e.g. [9, 81]. Although this works tolerably well for conventional modelling languages (i.e., those that do not use either clabjects or powertypes—see below), when applied to processes and methodologies several problems emerge (see discussion in [32]). Of these, two main issues prevail. Firstly, the double/multiple instanceOf relationship (as shown in Fig. 6) violates the language use and speech act theory determination that a concept cannot be an instance of another concept (where an instance is always part of a type-instance relationship and type is confounded with concept), e.g. [22, 80]. Secondly, instanceOf is a highly specific form of the more general "represents" relationship (Fig. 1). Furthermore, it is this "represents" linkage between model and reality that forms much of the focus of ontological

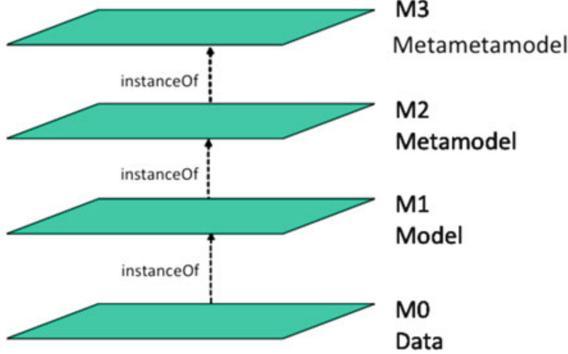


Fig. 6 OMG's theoretical four-layer hierarchy (modified slightly from [41]) © Pearson Education Limited

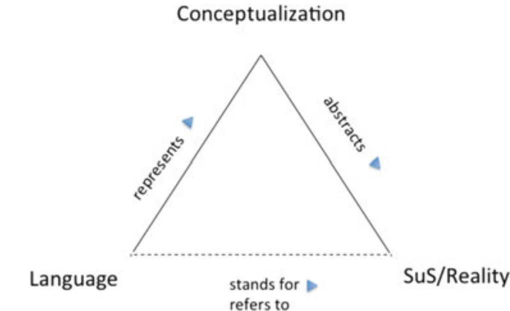


Fig. 7 Ogden and Richards' "meaning triangle" (a.k.a. Ullman triangle) for reality (or a specific reality domain such as banking or telecommunications, i.e., a Universe of Discourse, UoD)

thinking, as summarized, for instance, in [34], where Guizzardi reminds us of the Ullman triangle, originally conceived by Ogden and Richards [68].

Figure 7 depicts the Ullman triangle that expands on the single linkage of Fig. 1 (between model and SuS) by recognizing that the linkage between model (as a representation) and reality (the SuS) is indirect—there is a mediating effect of the human mind. This diagram also introduces the crucial mapping of "abstracts", which is discussed further in Sect. 3 below. Colloquially, it can be said that abstraction maps between a detailed description and a less detailed description (i.e., some information is discarded in the mapping). This side of the triangle can lead us to either token models or type models [54]—only the latter being of relevance to our study of DSMLs. Finally, Fig. 7 can be applied to a range of scales pertinent to the SuS; when the SuS has a specific domain focus, then the Language is a DSML, as noted earlier.

3 Mathematical Considerations

Here, we draw attention to the utilization and applicability of mathematics to models, meta-models and ontologies in software engineering with a specific focus on DSMLs. The approach taken is to base our mathematics on logic, in particular

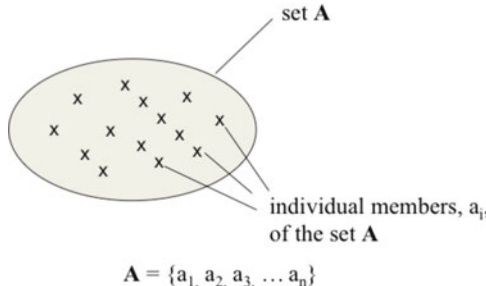


Fig. 8 A set and its members

on *propositional logic* (by which means a given statement can be unequivocally assessed as being either true (T) or false (F)) together with universal quantifiers, predicates, variables and functions, i.e., *first order predicate logic/calculus*. Here, we utilize set theory, noting that category theory may, in the future, provide a more comprehensive mathematical underpinning.

A set is a collection of individuals, usually called members (of the set) (see Fig. 8). Sets are often defined "by extension"—the standard *mathematical* definition, i.e., the explicit enumeration of each of its members.

If all the members of the set have a characteristic in common (defined by a predicate), the set can be also defined "by intension", the intension being a predicate that unequivocally characterizes the set elements. In other words, intensionality defines all possible members that legitimately belong to that set whereas extensionality enumerates the actual members that do belong to that set.

Definition by intension is highly relevant to the construction of MLs since it essentially defines the type, i.e., a type in software engineering has both an intension and an extension. Types represent concepts in the SuS, wherein the extension and intension are related by

$$\varepsilon\,(\mathrm{C}) = \{\mathrm{x} \mid \mathrm{p}\,(\mathrm{x})\}\,,\ \text{where } \mathrm{p} = \mathrm{i}\,(\mathrm{C}) \tag{1}$$

([56], Eq. (2)).

Set theory also provides us with definitions of membership and set inclusion ([21], pp. 14–15), which underpin our modelling concepts of instantiation and generalization, respectively, i.e.:

$$\text{(instantiation)} \qquad a_i \in A \text{ for } i = 1 \text{ to } n \tag{2}$$

such that a_i represents the instances and **A** the class. When a predicate is added to the class we get a statement for type conformance as:

$$\forall \mathrm{x},\ (\mathrm{x} \in \mathbf{A}) \Leftrightarrow \mathrm{p}\,(\mathrm{x}) \tag{3}$$

where p(x) is a property (predicate) of the element.

$$\text{(generalization) } \mathbf{A} \subseteq \mathbf{B} \tag{4}$$

which can also be understood as meaning:

$$\forall \text{x}, \ (\text{x} \in \mathbf{A}) \Rightarrow (\text{x} \in \mathbf{B}) \tag{5}$$

(for more mathematical detail, see [39]).

Functions, or mappings, between a pair of sets are also highly relevant to modelling. In particular, the abstraction mapping (for example, classification, generalization, aggregation) is seen as important for the creation of models, e.g. [60]. Roughly speaking [28], abstraction allows one to pay attention to the most relevant properties of the problem whilst discarding those that are considered less relevant[1] (see also, for example, [50]). Abstraction can thus be described through three complementary aspects:

1. Mapping: abstraction maps a representation[2] of the problem to a new, simpler representation. (This is the essence of modelling).
2. Simplification: by throwing away irrelevant details, the result of the abstraction process provides a simpler problem to be solved than the original.
3. Application: by preserving relevant, desirable properties, the solution to the simpler problem can be transferred and applied to the original problem.

Since, as noted above (property 2), an abstraction always discards detail, this may result in several entities in the SuS being mapped to a single one in the model ([2], p. 31). This loss of detail creates a simpler system from the SuS for the purposes of understanding the original SuS, e.g. [27, 28].

Classification and generalization can both be considered as kinds of abstractions, e.g. [56], since they both fit the three-aspect definition described above in terms of mapping, simplification and application. Classification refers to the allocation of members to a set defined by intension. Since Eq. (1) links the extensional and intensional definitions of a set, we can answer the question regarding whether or not a specific instance, e, belongs to the set C; in other words whether e is classified by concept C. From an extensional viewpoint, we have:

$$\text{e } \Gamma \text{C} \Leftrightarrow \text{e} \in \varepsilon\,(\text{C}) \tag{6}$$

[1] Some authors use the term "generalize" to mean ignore details (see, for example, [2], p. 40). Here, the term "generalize", and particularly "generalization", is used in the object-oriented sense of a relationship between a type and its super-type (or, equally, between a subtype and a type).

[2] Thus confounding the abstraction and representation links depicted in the "meaning triangle" (Fig. 7).

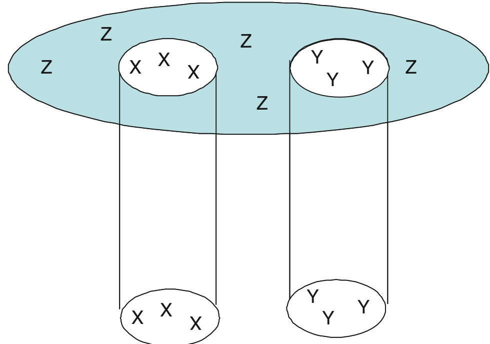


Fig. 9 Venn diagram representation of the generalization relationship (after [39])

where ε(C) is given intensionally by Eq. (1) and Γ is the symbol for the classification abstraction which is a non-transitive relationship. This parallels the set-member relationship given in Fig. 8. Generalization, on the other hand, denotes the set-theoretic relationship between a set **B** and a subset **A** (Eq. (4)) where we can write (in terms of extensions):

$$\varepsilon\,(\mathbf{A}) \subseteq \varepsilon\,(\mathbf{B}) \tag{7}$$

([56], Eq. (3)). We need to note that the generalization (subtyping) relation is transitive, since it involves the subsetting notion of Fig. 9, such that a member of a subset is also a member of each inclusion set.

The mapping between a set and a set member is the instantiation, or instanceOf, relation. Its inverse maps several instances to one set; where the set is actually a concept (Fig. 1) and thus Eq. (1) holds, this inverse being the classification relationship of Eq. (6).

It is also self-evident that since properties $p(x)$ are defined for a set, then each member of the set must possess the property. A consequence of this is non-transitivity for the instanceOf relation, which links type instances to the type.

Type-instance semantics are said to underpin much of current thinking in modelling and meta-modelling, e.g., [9]. In a modelling language like UML, however, this type-instance pattern is essentially applied twice: between M2 (as type) and M1 (as instance) and between M1 (as type) and M0 (as instance).[3] The entities in the M1 layer are thus sometimes viewed as types (classes in UML) (e.g., by modellers) and at other times viewed as instances (e.g., by meta-modellers). Translating this into a set representation has serious repercussions for the solidity and validity of the OMG strict-meta-modelling four-layer architecture, e.g., [39, 43], since it leads to M2-level sets, the members of which are themselves sets. Mathematical set theory supplies the notion of a "family of sets", known in software engineering as a "powertype" [29, 65]. A "powertype" is defined as a class the instances of which

[3]We will not discuss here a third possible application between M3 and M2.

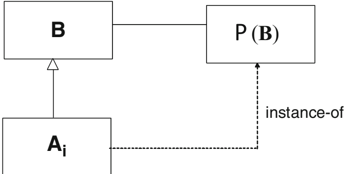


Fig. 10 Powertype pattern (after [39])

are subtypes of another class, itself related by means of a classification relationship to the powertype. In mathematics, there is a similar construct, that of the powerset, which is the set of all subsets (including the set itself and the empty set) and is usually written as P(**B**) or 2^B (since it has 2^B elements):

$$\mathrm{P}\left(\mathbf{B}\right) = \{\mathbf{A}|\mathbf{A} \subseteq \mathbf{B}\} \tag{8}$$

The elements, $\mathbf{A_i}$, of this powerset thus form a partition of the superset (**B**) (Fig. 10). In addition, each set (class) $\mathbf{A_i}$ is an instance of the set P(**B**), which, in turn, provides a partitioning rule for set **B**.

From a semantic perspective, a powertype of a type **B** (P(**B**) in Fig. 10, for example) represents an entity that can be described as "type of B", "kind of B" or some other similar classification over **B**. For example, a textbook example of powertype of the class Tree is the class TreeSpecies, since TreeSpecies classifies Tree. According to the definition of powertype, instances of TreeSpecies (the powertype) should be subtypes of Tree (the partitioned type): sample instances ($\mathbf{A_i}$ in Fig. 10) such as OakTree, PineTree or MapleTree are, indeed, instances of TreeSpecies (the oak tree species is a particular tree species) and also subtypes of Tree (oak trees are a particular type of trees). Note that the concepts of "tree" and "tree species", albeit closely related, are very different, as are those of an oak tree (one particular tree) and the oak tree species (corresponding to the biological construction of species).

We should also note that the instantiation relationship is often written as *is-a*; but this is ambiguous as many authors have pointed out, e.g., [7, 16]. Three meanings were identified in [7]:

- Is an instance of
- Is a kind of

and, specifically in UML [70], the most commonly used modelling language in software engineering

- A stereotype label used to "brand" a class in a model with another class [8].

This third interpretation is strictly a misuse of the UML's modelling rules, e.g., [40], and thus outside the scope of our current discussion. The second interpretation is highly relevant here since it represents the abstraction relationship more typically called generalization (or its inverse, specialization) which, in object-oriented programming languages, is usually implemented using the inheritance capability of the language.

Instantiation and classification are clearly related, as noted above. Strictly, classification provides a test regarding whether an individual satisfies the intension of a class (set). Instantiation, on the other hand, selects an individual from a pre-existing set or creates an individual by using the criteria given by the set's intension as a template and is therefore more relevant to the current discussion. Indeed, Abadi and Cardelli ([1], Sects. 2.1 and 3.1) note that the difference between type and class is that a type is simply about conformance to a signature (similar to defining a set by intension), whereas a class provides a templating mechanism, a "creation machine" capable of instantiation.

In summary, classification determines if an individual belongs to an intensionally defined set, instantiation relates individuals to sets (types) and generalization relates types to (super)types. A more detailed comparison of classification and generalization has been undertaken recently in [56]. A highly relevant question is which of these (if any) is useful in relating models to metamodels.

4 Models and Metamodels

As we have seen, a model is an abstraction of an SuS (Fig. 7). This modelling abstraction can be written [54, 55] as:

$$\rho\,(\mathrm{S},\mathrm{M}) \rightarrow \mathrm{S} \lhd \mathrm{M} \tag{9}$$

(i.e., ρ is a subset of ◁) where ρ is the representational relationship between model M and SuS S and ◁, the "modelled-by" (the representation relationship of Fig. 1) relationship, indicates a relationship between model and SuS that may be many-to-many. Seidewitz [81] states this as "a model is a representation of an SuS" and that "interpretation" is the inverse of "representation" (Fig. 1). In mathematical logics, interpretation is the assignment of semantic meaning (see also [47, 88]). Meaning is embedded in the particular representational choice and therefore the language used, e.g., [80]. Interpretation of a model is thus a mapping of the model's elements to the SuS elements that thus also permits model validation (with respect to the SuS) [81]. Interpretation is important as it is a mapping from the model elements to "reality" (the SuS) and thus provides a means whereby to check the correctness of the model.

Although in much of contemporary software engineering, which is currently heavily influenced by the OMG architecture of Fig. 6, modelling and meta-modelling are regarded as absolutes (e.g. "meta-model" implies M2, *always*), it should be noted that in fact the prefix "meta" is relative rather than absolute. According to [15], a model conforms to its meta-model (labelled "sem" in Fig. 11 since the metamodel defines the *sem*antics of the model) when all of its classes are instances of classes in the meta-model (labelled "meta"). Relationships in the model also need to be in compliance with the rules of the meta-model (see also [61]).

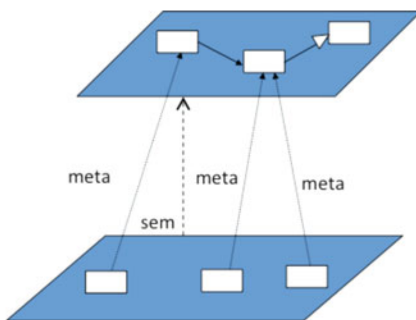


Fig. 11 A model has a "sem" relationship to its metamodel; elements in that model are related to elements in the meta-model by "meta" (a classification relationship) (after [39])

Joualt and Bézivin [48] propose a mathematical definition for *conformsTo* based on the notion of a so-called reference model.[4] A model, M, is said to conform to its reference model, ω, where M is a triplet (G, ω, μ) in which:

- $G = (N_G, E_G, \Gamma_G)$is a directed multi-graph consisting of a finite set of nodes, N_G, a finite set of edges, E_G, and a function $\Gamma_G : E_G \rightarrow N_G \times N_G$, i.e., mapping each edge to its source and its target nodes.
- ω is associated with a graph $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$
- There is a function μ that associates nodes and edges of G with nodes of G_ω given by

$$\mu : N_G \cup E_G \rightarrow N_\omega \tag{10}$$

Conformance is also addressed in [61] where a combination of category theory and graph transformation called Diagram Predicate Framework is introduced.

Powertype conformance (Fig. 10) can then be defined by extending Bézivin's [14] definition by enforcing Method Domain clabjects to map to both a powertype (an instance-of mapping) and a partitioned type (a generalization mapping).

Both interpretation and representational mappings as described in [81] relate *elements* of the SuS to *elements* of a model. Consequently, we can here equate these to the "meta" relationship of Bézivin (see Fig. 11). Since Seidewitz [81] notes that when a model is a representation of an SuS, all statements in the representation are true for the SuS under the interpretation mapping, we can conclude that the representational mapping is a more general case of the conformsTo mapping. In other words, the conformsTo mapping is a representational mapping in which the SuS is a model and the model is a meta-model.

[4]Note that this is a very different meaning from the use of "reference model" in software engineering standards published by ISO's JTC1 SC7.

5 DSMLs and Meta-Models

A modelling language is a means of expressing statements in the model ([81], p. 28). It is generally defined to consist of an abstract syntax and semantics, e.g. ([2], p. 13) and (arguably) pragmatics and concrete syntax (a.k.a. notation), e.g. [25, 49, 78]. Many authors (as we do here) exclude the notation from being an intrinsic part of a language, arguing that concepts are independent of any particular mode of representation. However, Silva Parreiras et al. [83] in contrast break down the syntax into (1) concrete, (2) abstract and (3) notation (graphical or lexical) and identify semantics as being either (a) formal (e.g., denotational) or (b) informal (e.g., natural language).

A DSML is then a modelling language that is constrained to the vocabulary of a particular (business) domain. As noted earlier, the (abstract) syntax of a DSML can be described in one of the several ways, e.g. by a meta-model ([33] Chap. 8, [58, 77]) or as a grammar (Fig. 4). A grammar represents a set of rules to illustrate how its basic elements, called the "alphabet", work and how "sentences" may be constructed from this alphabet. Others commonly use a meta-model to describe the abstract syntax of a modelling language, e.g. [58, 77], including DSLs ([33], p. 289 and 457). In addition, the scope of a DSML can be defined in terms of a domain ontology.

The semantics of a language give meaning to the symbols (i.e., alphabet) and hence to sentences, etc. The vocabulary of the language may be supplied by the ontology and/or the meta-model, e.g. ([3], p. 257). Guizzardi ([34], p. 36) describes this in terms of the "ontological commitment" of a given language; that is, a description of the specification of the conceptual model underpinning the language. This represents the world view or *Weltanshauung* embodied in the language. Without semantics, the appellation of "modelling language" is difficult to justify.

Whether general-purpose or domain-specific, a modelling language typically utilizes a meta-model to define the abstract syntax [25] plus separate semantics; or sometimes it is assumed that the meta-model *is* the modelling language (see [22] for a more detailed discussion of this apparent contradiction). Notation is sometimes included in the definition of an ML and sometimes not. Although not exclusively the case, meta-models do, however, figure prominently in any discussion of DSMLs.

A meta-model is an explicit specification of an abstraction, e.g. [15], expressed in a specific language. This is similar to Seidewitz's definition of a meta-model [81] as a specification model for a class of SuSs where the system now refers to a model (i.e., we apply Fig. 3 twice). In other words, each SuS is a valid model, i.e., "a meta-model makes statements about what can be expressed in the valid models of a certain modelling language" ([81], p. 28). It should be stressed that the abstraction needed here is a type abstraction since token models are not used for DSML construction.

Seidewitz [81] states that the UML standard is a meta-model and that, altogether, the (UML) meta-model includes notation, abstract syntax and semantics. Although it is common to represent a meta-model by using a graphical notation such as UML,

typically as a class diagram, there are other possibilities; for instance, Walter et al. [88] use text. Here, we have purposefully omitted notation from our definition since, for example, ISO/IEC 24744 [45] provides a meta-model without a notation[5] (see also [31, 33] Chap. 6]; or the meta-model may have several different alternative notations, e.g. [25, 57]. Kühne [54] argues that the definition of a meta-model might in the future be extended to include all these things.

While the definition, oft-quoted, of a meta-model as a "model of models" (or "a model of models expressed in a given language" [58]) is useful, an additional constraint is that all modelling activities must use type models rather than token models since we need to exclude transitivity, e.g., ([54], p. 378). Kühne [53] offers a useful heuristic to test the appropriateness of the appellation "meta-model": "Are the instances of their models *not* instances of them?"

Modelling and meta-modelling terminology can per se also be initially confusing. In the OMG architecture, UML belongs to M2 whereas a UML model belongs to M1 since "UML model" is short for "model that conforms to UML" or "model expressed using the UML". Similarly, while MOF belongs to M3, an MOF model is an M2 meta-model conformant to MOF (M3)—Guizzardi and Wagner [35] use the terms MOF model and also "MOF as a meta-modelling language" (to parallel the statement that UML is a modelling language). However, other authors use different and essentially contradictory terms to describe the M3-M2 connection. For example, Colomb et al. [18] describe an M2 meta-model that is conformant to MOF as an "MOF meta-model". Care must therefore be taken in assigning Mx layers across various papers in the literature. Here, we will use the terminology as utilized also in [35], i.e., an M3 model is an M2 artefact; an M2 model is an M1 artefact; an M2 artefact is both a meta-model and a modelling language; etc.

Layered architectures require a relation or mapping between pairs of layers. As noted in Sect. 2, the OMG's use of instanceOf between layers is easily confused with the intra-layer instantiation relationship. Bézivin [13] and Gašević et al. [26] emphasize that the two basic relationships for meta-modelling (i.e., multilayer modelling) are *representedBy* (as in Fig. 1) and *conformsTo*—paralleling the two basic OO modelling relationships of *instanceOf* and *inherits*.

5.1 Notational Considerations

As noted earlier, it is easy to confuse a model (concepts/statements) with the visual representation (or "visualization") of that model (e.g., class diagrams, differential equations, graph) [31]. Clarification can be gained by reconsidering Fig. 7 and focussing on the "*depicts*" relationship. Figure 7 states that a conceptualization is depicted by the use of a (modelling) language. Typically, the visualization of that

[5] A notation for ISO/IEC 24744 was added later [46]—see Sect. 6.1.

language is graphical. We thus explore some of the advice in the literature on the construction of symbols for use in a software-focussed modelling language.

A number of authors have offered advice on the key aspects of a good quality modelling language. Rumbaugh [76] suggests that the language should have the following characteristics:

1. Clear mapping of concepts to symbols
2. No overloading of symbols
3. Uniform mapping of concepts to symbols
4. Easy to draw by hand
5. Looks good when printed
6. Must fax and copy well using monochrome images
7. Consistent with past practice
8. Self-consistent
9. Distinctions not too subtle
10. Users can remember it
11. Common cases appear simple
12. Suppressible details.

Whilst this list is useful, it shows no influence from semiotics or usability studies. Indeed, when it was argued during an OMG meeting in Austin that the embryonic UML should be subject to pre-standardization usability tests, the proposal was not permitted to be put to the committee. Consequently, several UML symbols and annotations fail when subject to quality assessments, e.g. [63], and, furthermore, that the lack of provision of any design rationale for UML suggests that "this is acceptable practice even for the industry standard language" ([62], p. 757).

Constantine and Henderson-Sellers [19, 20] argue for the need to base symbols to be used in a modelling language on semiotic principles [73], differentiating between indexical signs (directly connected to the referent by physical association), iconic signs (having a likeness to the referent) and symbolic signs (having a connection by convention only). These ideas have been formalized more recently in [62], where it is argued that, although there are many goals that a notation could aim to encapsulate (e.g., simplicity, expressiveness, naturalness), the primary dependent variable for evaluating and comparing visual notations is cognitive effectiveness. Moody argues that cognitive effectiveness is not an emergent property but one that must be designed into the notation [59]. He notes that the representational *form* (i.e., the notational symbol) for a concept has equal, or even greater, influence on cognitive effectiveness than its content (including the underlying semantics) (see also [82])—as confirmed by empirical studies (as referenced in [63]).

Moody [62] provides nine principles for designing effective visual notations (Table 1). Semiotic clarity (Principle 1) requires a one-to-one correspondence between construct and symbol. If this is not achieved, "errors" such as symbol redundancy (several symbols used for one concept), overload (maps to more than one concept), excess (maps to no concept) or deficit (a concept has no symbolic representation) can occur (see [71] for an analysis of UML from this viewpoint). Symbol discriminability (Principle 2) is enhanced by ensuring that shapes likely to

Table 1 Principles for designing effective visual notations

Name of principle	Characteristics of principle
1. Semiotic clarity	Ensure there is a 1:1 correspondence between semantic constructs and graphical symbols
2. Perceptual discriminability	Clearly distinguish between different symbols
3. Semantic transparency	Use visual representations whose appearance suggests their meaning
4. Complexity management	Include explicit mechanisms for dealing with complexity
5. Cognitive integration	Include explicit mechanisms to support integration of information from different diagrams
6. Visual expressiveness	Use the full range and capacities of visual variables
7. Dual coding	Use text to complement graphics
8. Graphical economy	Ensure the number of different graphical symbols is cognitively manageable
9. Cognitive fit	Use different visual dialects for different tasks and audiences

Table 2 Eight characteristics of symbols of Bertin [11] from which an individual symbol can be constructed

Variable kind	Variable
Planar	Horizontal position; vertical position
Rational	Shape; size; colour; brightness; texture; orientation

be juxtaposed are from different "families" (e.g., curvilinear, polygonal), possibly also with additional use of colour or labels. Semantic transparency (Principle 3) uses cues to meaning, especially iconic signs rather than symbolic ones [19] and/or mnemonics. Good notational sets deal well with complexity (Principle 4) for which there are several options, although often none are used [63]. Various forms of decomposition are possible especially varying the abstraction levels by means of generalization/specialization and meronymic structures. Principle 5 (Table 1) encourages the use of systems of diagrams rather than one single diagram. Visual expressiveness (Principle 6) recommends the use of the large number of visual variables. Symbols can be differentiated in terms of eight variables [11, 62]—see Table 2. Use of colour when possible (e.g., not for printouts on B/W printers) can enhance the speed of recognition by a factor of three [85]. Text can also be useful as a complement to graphics (Principle 7). Cognitive limits to the number of symbols are recognized in Principle 8, especially problematical for novices [64]. Finally, Principle 9 focusses on the application of cognitive fit theory, e.g. [87], to visual software engineering notations. Cognitive differences exist between novices and experts such that a notation should have different subsets or dialects for such widely different user types.

Moody [63] notes that the application of the nine principles of Table 1 can lead to some problems resulting from interactions between principles, possibly leading to the need to make trade-offs or, conversely, to benefit by selecting identified synergies.

The symbols used in the notational package are thus clearly of primary importance. These symbols are sometimes called the "primary notation" in contrast to the "secondary notation" that adds further visual clues that aid in comprehension (in particular, differentiating between novices and experts [74]) whilst leaving the semantics unchanged [79]. These clues, which are called aesthetic guidelines in [85], include overall layout of model elements (investigated for ER diagrams in [37]), particularly focussing on line crossings, visual distance and back pointers [79]. Although clearly valuable, such concerns are outside the scope of our initial design of a DSML (here for agent-oriented system design).

6 Case Studies

As illustration of the application of these mathematically underpinned modelling and meta-modelling techniques, in this section we outline two DSMLs that we have developed in this formal mathematical framework. Section 6.1 describes the meta-model and notation developed as the ISO/IEC International Standard 24744 [45, 46]—also known as SEMDM (Software Engineering Meta-model for Development Methodologies). The mathematics specifically utilized in this is that of the powertype (mathematically a "family of sets")—(8) and Fig. 10.

The second case study is of an agent-oriented modelling language presented as a meta-model [12]. Although powertypes were not explicitly included, opportunities to do so are currently being explored. Section 6.2 presents the existing four-element meta-model together with elements of a proposed concrete syntax (notation).

6.1 Domain-Specific Modelling Language for Software Development Methodologies

As noted earlier, the need to standardize some aspects of the methodology for use in design and other aspects for process enactment cannot be satisfied with the architecture of Fig. 6. Instead, three domains are defined (Endeavour, Method, Meta-model as shown in Fig. 5) to reflect practice rather than the artificial structure of Fig. 6. Figure 12 shows how the powertype pattern of Fig. 10 is utilized in this new three-domain architecture. It should be noted that since the RequirementsSpecificationDocument class (in the model domain) has not only an instantiation relationship to a class in the meta-model (DocumentKind) but also a specialization relationship to a second class in the meta-model (Document), this architecture cannot be equated to the OMG four-layer architecture of Fig. 6 since the latter's use of strict meta-modelling forbids any relationship except instantiation to cross inter-level boundaries. Furthermore, this newer architecture (Fig. 5) provides a solution to a long-standing problem of enactment when discussing methodologies

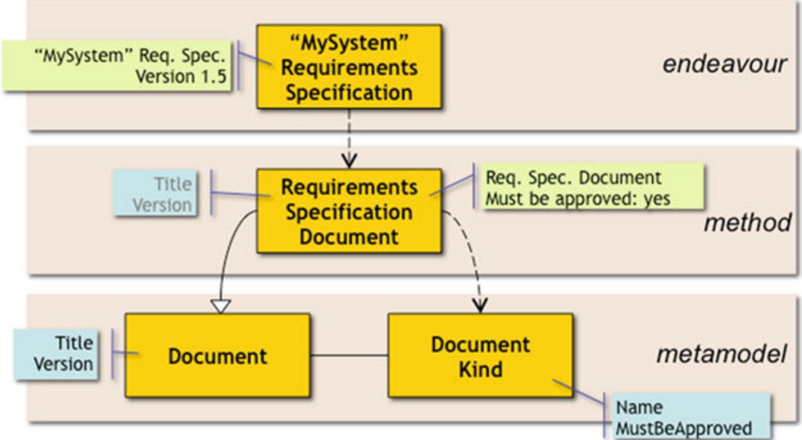


Fig. 12 The three domains used in ISO/IEC 24744 (after [38])

for software development. Strict meta-modelling is unable to transmit attribute values to level M0 when they are defined at M2 (without using a work-around such as potency, e.g., [6]). In contrast, in Fig. 12, attributes that refer to the method domain (the process model or methodology) are defined in the xxxKind class in the meta-model. Attributes that cannot be given a value until the endeavour domain need to be allocated to the xxx class in the meta-model; these are then inherited without change by the clabject in the method domain and then, in the endeavour domain, given a value pertinent to that particular project/endeavour.

Figure 12 depicts just one of the powertype patterns (Document-DocumentKind). The full ISO/IEC 24744 meta-model contains seven such patterns (known as "templates") as well as five regular meta-model classes (known as "resources"). In fact, the SEMDM meta-model can also be regarded as an ontology in the domain of methodology modelling [30].

Each of the templates and resources in Fig. 13 effectively defines a "family" of elements defined not only by each pattern but also by any subtype. In creating a notation for ISO/IEC 24744, semiotic principles were applied (as discussed above), taking into account also the need to be able to hand-draw the symbols (as opposed to the need to use a tool to draw a similarly-focussed notation like that of SPEM [69]) and the need to create a set of culturally independent shapes. Although colour is used by allocating a very specific hue to *all* members of a family, all symbols are equally recognizable by means of the family-focussed shape (e.g., rectilinear, curvilinear). Colour, when available, offers an added "clue" to the visualization but is not vital to understanding [84]. The design of this notation also takes into account all the semiotic advice discussed in Sect. 5.1. In particular, we have embodied Principles 1, 2 and 8 and to some degree Principles 3 and 6. Complexity management (Principle 4) and cognitive integration (Principle 5) only apply when diagram types rather than single symbols are discussed (see also [84]).

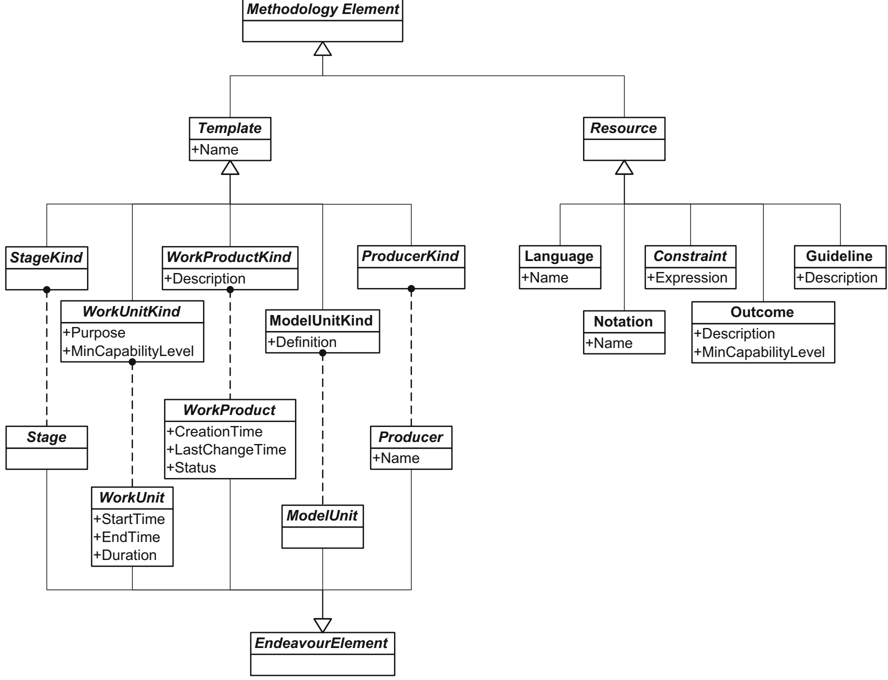


Fig. 13 The core of the ISO/IEC 24744 meta-model (after [84])

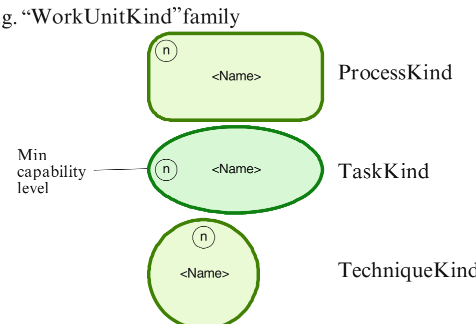


Fig. 14 Symbols for members of the WorkUnitKind family in ISO/IEC 24744

Finally, whilst supporting the ideas of Principle 9, empirical evidence resulting from extensive use of the proposed FAML notation (see Sect. 6.2 for details) is a prerequisite for future improvements.

Figure 14 illustrates the application of these semiotic principles by examples from the WorkUnitKind family of concepts in ISO/IEC 24744. There are three subtypes in the SEMDM meta-model: ProcessKind, TaskKind and TechniqueKind.

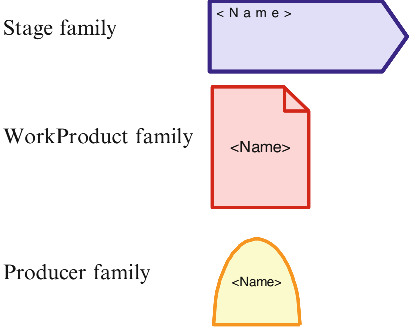


Fig. 15 ISO/IEC 24744 symbols showing one example from each family group of StageKind, WorkProductKind and ProducerKind

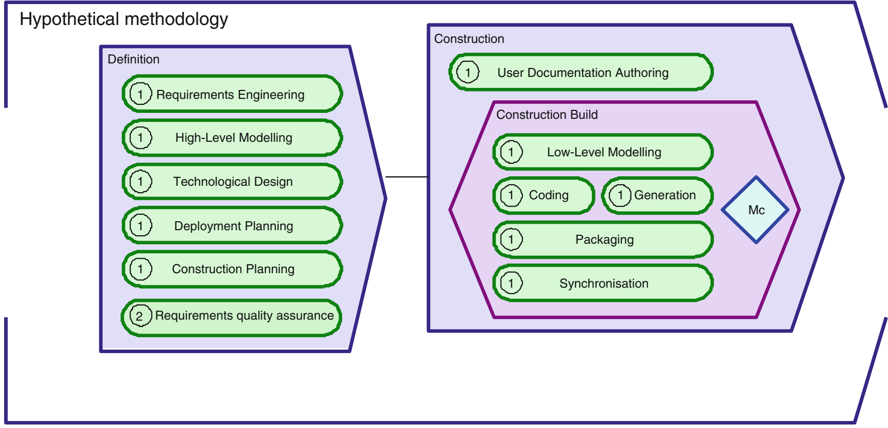


Fig. 16 Example lifecycle diagram drawn using the ISO/IEC 24744 notation (after [32])

All three are green and have a curvilinear shape. Their name is inside the shape as is a label (in a circle) to show the minimum capability level at which this operates [44]. In contrast, members of the Stage family (Fig. 15) are rectilinear with more than four sides (colour blue), members of the WorkProduct family are vertical, pink rectangle-based and members of the Producer family use the "mask" shape (originally used in the OPEN Modelling Language (OML) [24]) but with different orientations and yellow in colour.

A final and important consideration is to ensure that, when necessary, symbols can be superimposed or contained within each other—both in terms of shape, textual content and colour. For example, Fig. 16 depicts an example lifecycle diagram in which a time cycle kind (the "Hypothetical Methodology") consists of a number of process kinds that are shown inside the Definition phase kind. This is then followed by the Construction phase kind, which itself consists of a single process kind and

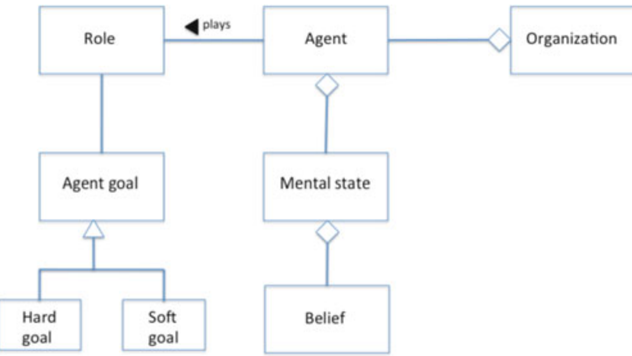


Fig. 17 Small part of the FAML metamodel

a build kind (labelled here as the Construction Build). This build kind consists of a number of process kinds and a milestone kind (blue diamond shape). Analysis in [84] confirms that the colours, when used, can be readily discriminated and that the shapes sit well within each other.

6.2 *Domain-Specific Modelling Language for Agents*

In a second example, we examine a DSML designed specifically for use in agent-oriented systems. FAML (FAME Agent-oriented Modelling Language) [12] is a modelling language defined initially as a meta-model (Fig. 17) to express the concepts required to construct an agent-oriented model, i.e., a work product in the ISO/IEC 24744 sense. In the FAML meta-model, an agent belongs to an organization (of agents), e.g., [66, 67], and, internally, has a so-called mental state, which consists of beliefs [91] (these being the agent/AI counterparts to human characteristics). Perhaps most importantly, agents play roles [67, 90]. Each role is associated with a specific (agent) goal, where goals are subdivided into hard and soft goals following Yu [92]. Hard goals model functional requirements and they are formulated so that it is possible to determine whether or not they have been satisfied; in contrast, soft goals are used to represent non-functional requirements, which typically can only be satisfied (i.e., fulfilled to an acceptable threshold) since they do not have well-defined binary achievement criteria [17].

As with SEMDM, we seek "families" of concepts to which we can ascribe similar visualizations. However, unlike SEMDM, the concepts in FAML are not, in general, related by generalization such that the creation of "families" by corralling types and all their subtypes, as done for SEMDM, is not appropriate here. Instead we seek to identify concepts that are focussed on a similar characteristic, e.g., work products and behaviour.

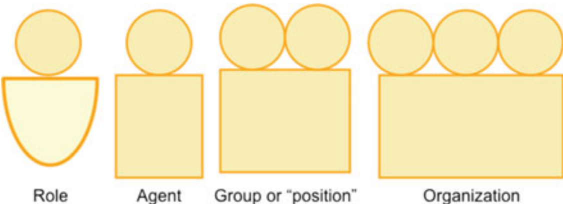


Fig. 18 Notation proposed for individual agents and groups of interacting agents (Groups are not used directly in FAML but are included for consistency with other AOMLs such as the notation used in INGENIAS) (after [42])

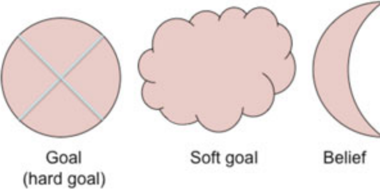


Fig. 19 Notation proposed for goals and beliefs (after [42])

We illustrate that ML notational design with two small examples—for full details, see [42]. Figure 18 depicts agents and the roles they play. Agents are also generally regarded as "social" and so we also need graphical support for more extensive groupings of individual agents. In FAML, the concept of an organization is supported (Fig. 17) but not the concept of "group". However, much of this notation is borrowed from INGENIAS [72] so we include a notation for group—especially since FAML aims to provide a common, unified agent notation rather than a methodologically singular one.

As noted above, agents have goals, which may be hard goals or soft goals. They also have beliefs, especially when the BDI internal agent architecture [75] is adopted. We choose symbols that are all curvilinear (Fig. 19) and of the same colour and are aligned as far as possible with current usage—here soft goals from Tropos [17].

7 Conclusions and Future Work

In the context of DSMLs, we have studied the relationships between models, meta-models, modelling languages and ontologies, which are not well defined in the literature. In particular, the implications of the strict meta-modelling paradigm fostered by the OMG in relation to the type/instance duality are often described in a vague and equivocal fashion. In this chapter we provide a solid theoretical foundation for the construction of DSMLs that can help define both the abstract

and concrete syntax aspects. Two example languages are described: the Software Engineering Meta-model for Development Methodologies (ISO/IEC International Standard 24744) and FAML (a language for the specification of agent-oriented software systems).

We have also briefly considered the mathematics relevant to DSMLs. In particular, we have referenced set theory as one means by which to create a mathematical underpinning that links together meta-models to DSMLs. Understanding and appreciating these formal mathematical linkages and descriptions for a DSML will help adopters and modellers to create good quality models. In particular, a DSML (as any other ML) should have a firm meta-model and ontological (preferably a foundational ontological) basis.

In particular, powertypes have been introduced as an alternative to the OMG strict meta-modelling approach, which can be especially helpful when modelling complex situations in domain-specific settings that require the representation of an entity plus the kinds of that entity in the same model.

Secondly, we note that a formally defined DSML needs a means of communication, typically by means of the addition of a visualizing notation a.k.a. concrete syntax. Here, we have presented some of the semiotic ideas in the literature.

We have illustrated some of these formal ideas with two case studies. We have discussed briefly the meta-model and notation embodied in ISO/IEC 24744, which uses powertypes to represent method-level (powertype) and endeavour-level (partitioned type) entities in the same model. Secondly, we have illustrated the main elements of the notation and its relation to the underpinning meta-model for a new agent-oriented modelling language based on the FAML meta-model.

Further work is envisaged in the formal sense to consolidate and extend the mathematical underpinning for DSMLs—and encouraging an appreciation of its need amongst practitioners—together with testing and likely further refinement of the notation for FAML.

Acknowledgements We wish to thank Mats Lind and Haris Mouratidis for their comments on the FAML notation. BH-S also wishes to acknowledge the support of the Australian Research Council through grant DP0878172. This is contribution 12/01 of the Centre for Object Technology Applications and Research within the Human Centred Technology Design centre at the University of Technology, Sydney.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, New York (1996)
2. Alagar, V.S., Periyasamy, K.: Specification of Software Systems. Springer, Berlin (1998)
3. Aßmann, U., Zschaler, S., Wagner, G.: Ontologies, meta-models, and the model-driven paradigm. In: Calero, C., Ruiz, F., Piattini, M. (eds.) Ontologies for Software Engineering and Software Technology, pp. 239–273. Springer, Berlin (2006)
4. Atkinson, C.: Metamodelling for distributed object environments. In: First International Enterprise Distributed Object Computing Workshop (EDOC'97). Brisbane (1997)

5. Atkinson, C.: Supporting and applying the UML conceptual framework. In: Bézivin. J., Muller, P-A. (eds.) The Unified Modeling Language. «UML» 1998: Beyond the Notation. LNCS, vol. 1618, pp. 21–36, Springer, Berlin (1998)
6. Atkinson, C., Kühne, T.: The essence of multilevel metamodelling. In: Gogolla, M., Kobryn, C. (eds.) «UML»2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools. LNCS, vol 2185, pp. 19–33, Springer, Berlin (2001)
7. Atkinson, C., Kuhne, T., Henderson-Sellers, B.: To meta or not to meta—that is the question. JOOP/ROAD **13**(8), 32–35 (2000)
8. Atkinson, C., Kuhne, T., Henderson-Sellers, B.: Systematic stereotype usage. Software. Syst. Model. **2**(3), 153–163 (2003)
9. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. IEEE Trans. Software. Eng. **35**(6), 742–755 (2009)
10. Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving extensibility through product-lines and domain-specific languages: a case study. ACM Trans. Softw. Eng. Methodol. **11**(2), 191–214 (2002)
11. Bertin, J.: Semiology of Graphics: Diagrams, Networks, Maps. University of Wisconsin Press, Madison (1983)
12. Beydoun, G., Low, G., Henderson-Sellers, B., Mouratidis, H., Gomez-Sanz, J., Pavon, J., Gonzalez-Perez, C.: FAML: a generic metamodel for MAS development. IEEE Trans. Softw. Eng. **35**(6), 841–863 (2009)
13. Bézivin, J.: In search of a basic principle for model-driven engineering. Upgrade **V**(2), 21–24 (2004)
14. Bézivin, J.: On the unification power of models. Softw. Syst. Model. **4**, 171–188 (2005)
15. Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: Presented at ASE'01, Automated Software Engineering, San Diego, 26–29 November 2001. Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01), p. 273. IEEE Computer Society Press, Coronado (2001)
16. Bézivin, J., Lemesle, R.: Ontology-based layered semantics for precise OA&D modeling. In: Bosch, J., Mitchell, S. (eds.) Object-Oriented Technologys: ECOOP'97 Workshop Reader. LNCS, vol. 1357, pp. 287–292. Springer, Berlin (1998)
17. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopolous, J., Perini, A.: Tropos: an agent-oriented software development methodology. Autonomous. Agents. Multi-Agent. Syst. **8**(3), 203–236 (2004)
18. Colomb, R., Raymond, K., Hart, L., Emery, P., Welty, C., Xie, G.T., Kendall, E.: The object management group ontology definition metamodel. In: Calero, C., Ruiz, F., Piattini, M. (eds.) Ontologies for Software Engineering and Software Technology, pp. 217–247. Springer, Berlin (2006)
19. Constantine, L.L., Henderson-Sellers, B.: Notation matters: Part 1—framing the issues. Rep. Object. Anal. Des. **2**(3), 25–29 (1995)
20. Constantine, L.L., Henderson-Sellers, B.: Notation matters: Part 2—applying the principles. Rep. Object. Anal. Des. **2**(4), 20–23 (1995)
21. Denning, P.J., Dennis, J.B., Qualitz, J.E.: Machines, Languages, and Computation. Prentice-Hall, Englewood Cliffs (1978)
22. Eriksson, O., Henderson-Sellers, B., Ågerfalk, P.J.: Ontological and Linguistic Metamodelling Revisited—A Language Use Approach (2013, in press)
23. Falbo RdeA., Guizzardi, G., Duarte, K.C.: An ontological approach to domain engineering. In: Proceedings of International Conference on Software Engineering and Knowledge Engineering SEKE'02, ACM, Ischia, 15–19 July 2002
24. Firesmith, D., Henderson-Sellers, B., Graham, I.: OPEN Modeling Language (OML) reference manual. SIGS Books, New York, 276 pp (1997); Cambridge University Press, New York (1998)
25. Gargantini, A., Riccobene, E., Scandurra, P.: A semantic framework for metamodel-based languages. J. Automated. Softw. **16**(3–4), 415–454 (2009)
26. Gašević, D., Kaviani, N, Hatala, M.: On metamodeling in megamodels. In: Engels, G., et al. (eds.) MoDELS 2007. LNCS, vol. 4735, pp. 91–105. Springer, Berlin (2007)

27. Ghidini, C., Giunchiglia, F.: A semantics for abstraction. In: Lopez de Mantaras, R., Saitta L. (eds.) Proceedings of ECAI2004, pp. 343–352. IOS, Amsterdam (2004)
28. Giunchiglia, F., Walsh, T.: A theory of abstraction. Artif. Intell. **57**(2–3), 323–390 (1992)
29. Gonzalez-Perez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework. Softw. Syst. Model. **5**, 72–90 (2006)
30. Gonzalez-Perez, C., Henderson-Sellers, B.: An ontology for software development methodologies and endeavours. In: Calero, C., Ruiz, F., Piattini, M. (eds.) Ontologies in Software Engineering and Software Technology, pp. 123–152. Springer, Berlin (2006)
31. Gonzalez-Perez, C., Henderson-Sellers, B.: Modelling software development methodologies: a conceptual foundation. J. Syst. Software. **80**(11), 1778–1796 (2007)
32. Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley, Chichester (2008)
33. Greenfield, J., Short, K.: Software Factories. Wiley, Chichester (2004)
34. Guizzardi, G.: Ontological foundations for structural conceptual models, CTIT PhD Thesis Series, No. 05–74. Enschede, The Netherlands (2005)
35. Guizzardi, G., Wagner, G.: Towards ontological foundations for agent modelling concepts using the Unified Foundational Ontology (UFO). In: Bresciani, P., Giorgini, P., Henderson-Sellers, B., Low, G., Winikoff, M. (eds.) Agent-Oriented Information Systems II. LNAI, vol. 3508. pp. 110–124. Springer, Berlin (2005)
36. Harsu, M.: A survey on domain engineering. Report, Institute of Software Systems, Tampere University of Technology (2002)
37. Hay, D.C.: Data Model Patterns. Conventions of Thought. Dorset House Publishing Company, New York (1996)
38. Henderson-Sellers, B.: Method engineering: theory and practice. In: Karagiannis, D., Mayr, H.C. (eds.) Information Systems Technology and its Applications. Proceedings of the 5th International Conference ISTA 2006. Lecture Notes in Informatics (LNI) vol. P-84, pp. 13–23. Gesellschaft für Informatik, Bonn, 30–31 May (2006)
39. Henderson-Sellers, B.: On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages. SpringerBriefs in Computer Science. Springer, Heidelberg (2012)
40. Henderson-Sellers, B., Gonzalez-Perez, C.: Uses and abuses of the stereotype mechanism in UML1.4 and 2.0. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Model Driven Engineering Languages and Systems. 9th International Conference, MoDELS 2006, Genoa. LNCS, vol. 4199, pp. 16–26. Springer, Berlin (2006)
41. Henderson-Sellers, B., Unhelkar, B.: OPEN Modeling with UML. Addison-Wesley, London (2000)
42. Henderson-Sellers, B., Low, G.C, Gonzalez-Perez, C.: Semiotic considerations for the design of an agent-oriented modelling language. In: Bider, I. et al. (eds.) Proceedings of BPMDS 2012 and EMMSAD 2012. LNBIP, vol. 113, pp. 422–434. Springer, Heidelberg (2012)
43. Henderson-Sellers, B., Eriksson, O., Gonzalez-Perez, C., Ågerfalk, P.J.: Ptolemaic metamodelling? The need for a paradigm shift. In: Cueva Lovelle, J.M., Pelayo García-Bustelo, C., Sanjuán Martínez, O. (eds.) Progressions and Innovations in Model-Driven Software Engineering. IGI Global (2013, in press)
44. Henderson-Sellers, B., Serour, M., McBride, T., Gonzalez-Perez, C., Dagher, L.: Process construction and customization. J. Universal. Comput. Sci. **10**(4), 326–358 (2004)
45. ISO/IEC.: Software engineering—metamodel for software development. ISO/IEC 24744, Geneva (2007)
46. ISO/IEC.: 24744 Software engineering—metamodel for development methodologies annex A—notation. International Organization for Standardization/International Electrotechnical Commission, Geneva (2010)
47. Jackson, M.: Some notes on models and modelling. In: Borgida, A.T et al. (eds.) Mylopoulos Festschrift. LNCS, vol. 5600, pp. 68–81. Springer, Berlin (2009)
48. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Gorrieri, R., Wehrheim. H. (eds.) Formal Methods for Open Object-based Distributed Systems. LNCS, vol. 4037, pp. 171–185. Springer, Berlin (2006)

49. Karagiannis, D., Kuhn, D.: Metamodelling platforms. In: Bauknecht, K., Min Tjoa, A., Quirchmayer. (eds.) In: Proceedings of the 3rd International Conference EO-Web2002-Dexa2002. LNCS, vol. 2455, pp. 182–195. Springer, Berlin (2002)
50. Keet, M.: Enhancing comprehension of ontologies and conceptual models through abstractions. In: Basili, R., Pazienza, M.T. (eds.) AI*IA 2007. LNAI, vol. 4733, pp. 813–821. Springer, Berlin (2007)
51. Kelly, S., Pohjonen, R.: Worst practices for domain-specific modelling. IEEE. Softw. **26**, 22–29 (2009)
52. Kleppe, A.: A language description is more than a metamodel. Paper Presented at ATEM2007 (part of MoDELS2007), IEEE Online Publication (2007)
53. Kuhne, T.: What is a model? Dagstuhl Seminar Proceedings 04101. http://drops.dagstuhl.de/opus/volltexte/2005/23 (2005)
54. Kuhne, T.: Matters of (meta-)modeling. Softw. Syst. Model. **5**, 369–385 (2006)
55. Kuhne, T.: Clarifying matters of (meta-)modeling: an author's reply. Softw. Syst. Model. **5**, 395–401 (2006)
56. Kühne, T.: Contrasting classification with generalization. In: Kirchberg, M., Link, S. (eds.) In: Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modelling. Conferences in Research and Practice in Information Technology, 96, pp. 71–78. Australian Computer Society, Sydney (2009)
57. Kurtev, I., Bézivin, J., Joualt, F., Valduriez, P.: Model-based DSL frameworks. In: OOPSLA'06: Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 602–616, ACM (2006)
58. Laarman, A., Kurtev, I.: Ontological metamodelling with explicit instantiation. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE2009. LNCS, vol. 5969, pp. 174–183. Springer, Berlin (2010)
59. Larkin, J.H., Simon, H.A.: Why a diagram is (sometimes) worth ten thousand words. Cogn. Sci. **11**(1), 65–100 (1987)
60. Ludewig, J.: Models in software engineering—an introduction. Softw. Syst. Model. **2**, 5–14 (2003)
61. Mantz, F., Lamo, Y., Rossini, A., Wolter, U., Taentzer, G.: Formalising metamodel evolution based on category theory. In: Pettersson, P., Seceleanu, C. (eds.) Proceedings of the 23rd Nordic Workshop on Programming Theory. 26–28 October, 2011, pp. 73–75, Västerås, Sweden Technical Report 254/2011, Mälardalen University (2011)
62. Moody, D.L.: The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE. Trans. Softw. Eng. **35**(6), 756–779 (2009)
63. Moody, D.L., van Hillegersberg, J.: Evaluating the visual syntax of UML: an analysis of the cognitive effectiveness of the UML family of diagrams. In: Gaševic, D., Lämmel, R., Wyk, E. (eds.) Proceedings of the First International Conference on Software Language Engineering. LNCS, vol. 5452, pp. 16–34. Springer, Berlin (2009)
64. Nordbotten, J.C., Crosby, M.E.: The effect of graphic style on data model interpretation. Inf. Syst. J. **9**(2), 139–156 (1999)
65. Odell, J.J.: Power types. J. Object-Oriented. Prog. **7**(2), 8–12 (1994)
66. Odell, J.J., Parunak, H.V.D., Fleischer, M.: The role of roles in designing effective agent organizations.In: Garcia, A et al. (eds.) SELMAS 2002. LNCS, vol. 2603, pp. 27–38. Springer, Berlin (2003)
67. Odell, J.J., Parunak, H.V.D., Fleischer, M.: Modeling agent organizations using roles. Softw. Syst. Model. **2**, 76–81 (2003)
68. Ogden, C.K., Richards, I.A.: The Meaning of Meaning. Harcourt. Brace and World, New York (1923)
69. OMG.: Software & Systems Process Engineering Meta-Model Specification Version 2.0, formal/2008-04-01 (2008)
70. OMG.: OMG Unified Modeling LanguageTM (OMG UML), Superstructure. Version 2.4.1. formal/2011-08-06 (2011)

71. Opdahl, A., Henderson-Sellers, B.: Ontological evaluation of the UML using the Bunge-Wand-Weber model. Softw. Syst. Model. **1**(1), 43–67 (2002)
72. Pavón, J., Gómez-Sanz, J.J., Fuentes, R.: The INGENIAS methodology and tools. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 236–276. Idea Group Inc, Hershey (2005)
73. Peirce, C.S.: Collected Papers v. 2, paragraphs 243–63. written circa (1903)
74. Petre, M.: Why looking isn't always seeing: readership skills and graphical programming. Comms. ACM. **38**(6), 33–44 (1995)
75. Rao, A.S., Georgeff, M.P.: BDI agents: from theory to practice. Technical Note 56, Australian Artificial Intelligence Institute (1995)
76. Rumbaugh, J.: Notation notes: principles for choosing notation. J. Object Oriented. Prog. **9**(2), 11–14 (1996)
77. Saeki, M., Kaiya, H.: On relationships among models, meta models and ontologies. In: Gray, J., Tolvanen, J.-P., Sprinkle, J. (eds.) Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling. Computer Science and Information System Reports, Technical Reports, TR-37. University of Jyväskylä, Jyväskylä (2007)
78. Sánchez Cuadrado, J., Garcıa Molina, J.: A model-based approach to families of embedded domain specific languages. IEEE. Trans. Softw. Eng. **99**, 825–840 (2009)
79. Schrepfer, M., Wolf, J., Mendling, J., Reijers, H.A.: The impact of secondary notation on process model understanding. In: Persson, A., Stirna, J. (eds.) The Practice of Enterprise Modeling (Second IFIP WG 8.1 Working Conference, PoEM 2009, Stockholm, Sweden, 18–19 November, 2009, Proceedings), pp. 161–175. Springer, Berlin (2009)
80. Searle, J.R.: Speech Acts. An Essay in the Philosophy of Language. Cambridge University Press, Cambridge (1969)
81. Seidewitz, E.: What models mean. IEEE. Softw. **20**, 26–31 (2003)
82. Siau, K.: Informational and computational equivalence in comparing informational modelling methods. J. Database. Manag. **15**(1), 73–86 (2004)
83. Silva Parreiras, F., Staab, S., Winter, A.: On marrying ontological and metamodeling technical spaces. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Dubrovnik, 3–7 September 2007, pp. 439–448. ACM (2007)
84. Sousa, K., Vanderdonckt, J., Henderson-Sellers, B., Gonzalez-Perez, C.: Evaluating a graphical notation for modelling software development methodologies. J. Vis. Lang. Comput. **23**(4), 195–212 (2012)
85. Treisman, A.: Perception grouping and attention in visual search for features and for objects. J. Exp. Psychol. Hum. Percept. Perform. **8**(2), 194–214 (1982)
86. Vessey, I.: Cognitive fit: a theory-based analysis of the graphs versus tables literature. Decis. Sci. **22**, 219–240 (1991)
87. Walter, T., Silva Parreiras, F., Staab, S.: OntoDSL: an ontology-based framework for domainspecific languages. In: Schurr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 408–422. Springer, Berlin (2009)
88. Ward, C.B., Henderson-Sellers, B.: Utilizing dynamic roles for agents. J. Obj. Technol. **8**(5), 116–130 (2009)
89. Winikoff, M., Padgham, L., Harland, J.: Simplifying the development of intelligent agents. In: Proceedings of the 14th Australian Joint Conference on Artificial Intelligence (AI'01), Adelaide, 10–14 December (2001)
90. Yu, E.S-K.: Modelling strategic relationships for process reengineering. PhD Thesis, University of Toronto (1995)

Ontology-Based Evaluation and Design of Visual Conceptual Modeling Languages

Giancarlo Guizzardi

Abstract In this chapter, we present a framework for the evaluation and (re)design of modeling languages. In our approach, this property can be systematically evaluated by comparing a concrete representation of the worldview underlying the language (captured in the language's meta-model), with an explicit and formal representation of a conceptualization of that domain (a reference ontology). Moreover, we elaborate on formal characterizations for the notions of reference ontology, conceptualization and meta-model, as well as on the relations between them. By doing this, we can also formally define the relation between the state of affairs in reality deemed possible by an ontology and the grammatical models admitted by a modeling language. The precise characterization of this relation allows for a systematic improvement of a modeling language by incorporating ontological axioms as grammatical constraints in the language's meta-model. Furthermore, we demonstrate how an approach based on visual simulation could be used to assess this relation, i.e., to evaluate the distance between the *valid models* of a language and the *intended models* according to the underlying conceptualization. Finally, we demonstrate how the use of a system of formal ontological properties can be systematically exploited in the design of pragmatically efficient domain-specific visual languages.

Keywords Ontology • Visual languages • Language engineering

1 Introduction

The objective of this chapter is to discuss the design and evaluation of modeling languages for capturing phenomena in a given domain according to a conceptu-

G. Guizzardi (✉)
Ontology and Conceptual Modeling Research Group (NEMO), Federal University of Espírito Santo (UFES), Vitoria, Brazil
e-mail: gguizzardi@inf.ufes.br

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_13, © Springer-Verlag Berlin Heidelberg 2013

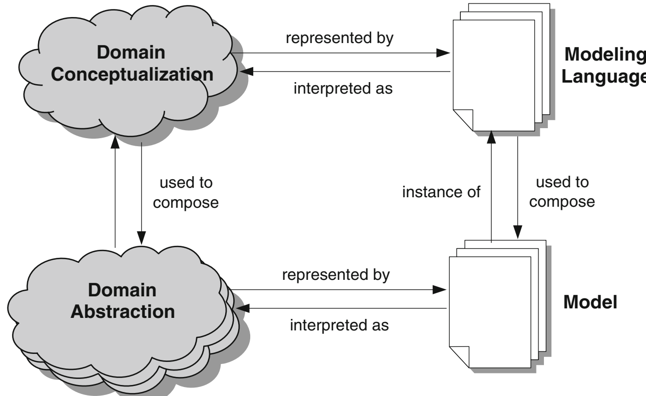


Fig. 1 Relations between conceptualization, abstraction, modeling language, and model

alization of that domain. In particular, we focus on two properties of a modeling language with respect to a given real-world domain [1]: (i) *domain appropriateness*, which refers to truthfulness of the language to the domain and (ii) *comprehensibility appropriateness*, which refers to the pragmatic efficiency of the language to support communication, understanding, and reasoning in the domain.

The elements constituting a *conceptualization* of a given domain are used to articulate abstractions of certain state of affairs in reality. We name them here *domain abstractions*. Domain conceptualizations and abstractions are intangible entities that only exist in the mind of the user or a community of users of a language. In order to be documented, communicated, and analyzed, these entities must be captured in terms of some concrete artifact, namely a *model*. Moreover, in order to represent a model, a *modeling language* is necessary. Figure 1 depicts the relation between a conceptualization, domain abstraction, model, and modeling language.

In this chapter, we elaborate on a framework that can be used to evaluate the suitability of a language to model a set of real-world phenomena in a given domain. In our approach, *domain* and *comprehensibility appropriateness* can be systematically evaluated by comparing the level of homomorphism between a concrete representation of the worldview underlying the language (captured in a *meta-model of the language*), with an explicit and formal representation of a conceptualization of that domain (a *reference ontology* [2]). Our framework comprises a number of properties that must be reinforced for an isomorphism to take place between these two entities. If an isomorphism can be guaranteed, the implication for the human agent who interprets a diagram (model) is that his interpretation correlates precisely and uniquely with an abstraction being represented. By contrast, in case the correlation is not an isomorphism there may be multiple unintended abstractions that match the interpretation.

The framework presented here builds on existing work in the literature. In particular, it considers the frameworks proposed in [3, 4], which focus on evaluating the match between individual diagrams and the state of affairs they represent, and the pioneering approach of Wand and Weber presented in [5, 6], which focuses on

the system of representations as a whole, i.e., a language. Although our approach is also centered in the language level, we show that, by considering desirable properties of the mapping of individual diagrams onto what they represent, we are able to account for desirable properties of the diagrams' modeling languages. In this way, we extend the original proposal presented in [5]. We also build here on the work of the philosopher of language H. P. Grice [7] and his notion of *conversational maxims* that states that a speaker is assumed to make in dialog contributions which are *relevant*, *clear*, *unambiguous*, and *brief*, *not overly informative* and *true according to the speaker's knowledge*. Furthermore, in comparison with [3, 4] and [5], by presenting a formal elaboration of the nature of the entities depicted in Fig. 1 as well as their interrelationships, we manage to present a more general and precise characterization of the characteristics that a language must have to be considered truthful to a given domain.

The rest of this chapter is structured as follows. Section 2 introduces the evaluation framework proposed here. Section 3 presents a formal characterization of the notions of reference ontology, conceptualization, and meta-model, as well as on the relations between these notions. By doing this, we can also formally define the relation between the state of affairs in reality deemed possible by a reference ontology and the grammatical models admitted by a modeling language. In Sect. 4, we exemplify the approach proposed by reporting on the design of an ontologically well-founded version of UML for the purpose of conceptual modeling and domain ontology engineering. This language (now termed *OntoUML*), in addition to an extensive case study of the approach discussed here, is itself a contribution to the engineering of domain-specific languages. This is discussed in depth in Sect. 5. Finally, Sect. 6 presents final considerations of the chapter.

It is important to highlight that this chapter can be considered as an extension of [1]. In particular, Sects. 4 and 5 represent a substantial extension to the original paper. Section 6 also contains a more systematic comparison with the works of Gurr and Wand & Weber.

2 Language and Conceptualization

The purpose of the current chapter is to discuss the design and evaluation of artificial modeling languages for capturing phenomena in a given material domain according to a conceptualization of this domain. Before targeting this at a language level, i.e., at a level of a system of representations, we start discussing the simpler relation between particular models and abstractions of portions of reality.

In [3, 4], Gurr presents a framework to formally evaluate the relation between the properties of a representation system and the properties of the domain entities they represent. According to him, representations are more or less effective depending on the level of homomorphism between the algebras used to represent what he terms the *representing* and the *represented* world, which correspond to the *model* and *domain abstraction* in Fig. 1, respectively.

Gurr argues at length that the stronger the match between a model and its representing diagram, the easier it is to reason with the latter. The easiest case is when these matches are *isomorphisms*. The implication of this for the human agent who interprets the diagram is that his interpretation correlates precisely and uniquely with an abstraction being represented. By contrast, where the correlation is not an isomorphism then there may potentially be a number of different models that would match the interpretation.

The evaluation framework proposed by Gurr focuses on evaluating the match between individual diagrams and the state of affairs (*abstractions*) they represent. In [5, 6], another framework is defined for evaluating *expressiveness* and *clarity* of modeling grammars, i.e., with the focus on the system of representations as a whole. In other words, in the latter proposal, the authors focus on the relation between what is named *Conceptualization* and *Modeling Language* in Fig. 1. In this chapter, these two proposals are merged into one single evaluation framework. We focus our evaluation on the level of the system of representations. Nevertheless, as it will be shown in the following subsections, by considering desirable properties of the mapping of individual diagrams onto what they represent, we are able to account for desirable properties of the modeling languages used to produce these diagrams, extending in this way Wand & Weber's original proposal.

It is important to highlight that in the proposal discussed here we want to systematically evaluate the level of homomorphism between *Conceptualization* and *Language* by comparing concrete representation of these entities: the notion of an Ontology as a concrete representation of a conceptualization is discussed in depth and formally characterized in Sect. 3; as a concrete representation of a language, we take the language meta-model. It is important to clarify, nonetheless, that by meta-model of the language we do not mean the actual description of the abstract syntax of the language. Instead what is meant here is what is termed in [2] the *Ontological Meta-model of the Language* or, simply, the *Ontology of the Language*. This meta-model is meant to capture the worldview underlying the language represented by the language modeling primitives. The definitive abstract syntax of the language is a language engineering artifact derived from that by considering a number of relevant nonfunctional requirements (e.g., to facilitate meta-model management or mapping to a particular implementation technology, decidability, and complexity in reasoning, etc.) [2].

In [3], four properties are defined, which are required to hold for a homomorphic correlation between a *represented world* and a *representation* to be an isomorphism: *lucidity*, *soundness*, *laconicity*, and *completeness* (Fig. 2). These properties are discussed as follows.

2.1 Lucidity and Construct Overload

A model M is called *lucid* with respect to (w.r.t.) an abstraction A if a (representation) mapping from A to M is *injective*. A mapping between A and M is injective iff

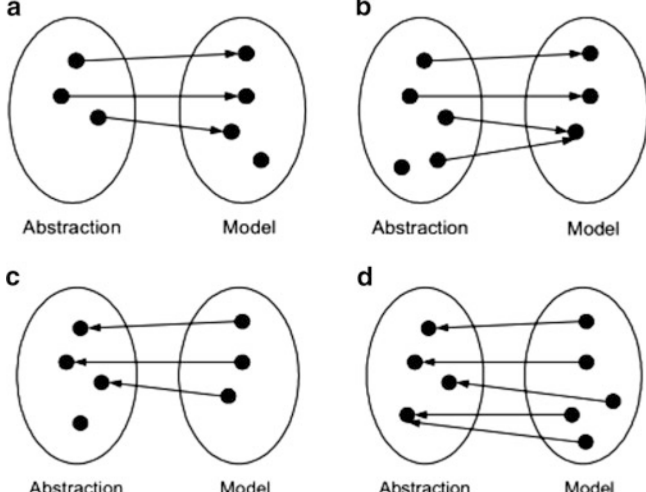


Fig. 2 Examples of lucid (**a**) and sound (**b**) representational mappings from Abstraction to Model; examples of laconic (**c**) and complete (**d**) interpretation mappings from Model to Abstraction

every entity in the model M represents *at most* one (although perhaps none) entity of the abstraction A. An example of an injective mapping is depicted in Fig. 2a.

The notion of lucidity at the level of individual diagrams is strongly related to the notion of *ontological clarity* at the language level as discussed in [6]. In that article, the author states that the ontological clarity of a modeling grammar is undermined by what he calls *construct overload*: *"construct overload occurs when a single grammatical construct can stand for two or more ontological constructs. The grammatical construct is overloaded because it is being used to do more than one job"*.

The notions of *lucidity* and *ontological clarity* albeit related are not identical. A construct can be overloaded in the language level, i.e., it can be used to represent different concepts, but every manifestation of this construct in individual specifications is used to represent only one of the possible concepts. Nevertheless, non-lucidity can also be manifested at a language level. We say that a language (system of representation) is non-lucid according to a conceptualization if there is a construct of the language which is non-lucid, i.e., a construct that when used in a model it stands for more than one entity of the represented abstraction. Non-lucidity at the language level is a special case of construct overload that does entail non-lucidity at the level of individual specifications.

Construct overload is considered an undesirable property of a modeling language since it causes ambiguity and, hence, undermines clarity. When it exists, users have to bring additional knowledge not contained in the specification to understand the phenomena that are being represented. In summary, a modeling language should not contain construct overload and every instance of a modeling construct of this language should represent only one individual of the represented domain abstraction.

2.2 Soundness and Construct Excess

A model M is called *sound* w.r.t. an abstraction A if a (representation) mapping from A to M is *surjective*. A representation mapping from A to M is surjective iff the corresponding interpretation mapping from M and A is total, i.e., iff every entity in the model M represents *at least* one entity of abstraction A (although perhaps several). An example of a surjective representation mapping is depicted in Fig. 2b.

Unsoundness at the level of individual specifications is strongly related to unsoundness at the language level, a property that is termed *construct excess* in [6]: *"construct excess occurs when a grammatical construct does not map onto an ontological construct"*. Although construct excess can result in the creation of unsound specifications, soundness at the language level does not prohibit the creation of unsound specifications. For instance, suppose a domain of natural numbers and a language that uses arrows to represent the *less-than* relation between natural numbers and labeled boxes to represent these numbers. Now, suppose we use this language to build a specification in which we have a box labeled X *arrow-connected* to the box representing the number 0. Although the language used is sound, i.e., all construct types have an interpretation in terms of domain types, the aforementioned specification produced using the language is unsound, given that there is no referent to the box labeled X in the domain. Since no mapping is defined for the exceeding construct, its meaning becomes uncertain, hence, undermining the clarity of the specification.

According to [6], users of a modeling language must be able to make a clear link between a modeling construct and its interpretation in terms of domain concepts. Otherwise, they will be unable to articulate precisely the meaning of the specifications they generate using the language. Therefore, a modeling language should not contain construct excess and every instance of its modeling constructs must represent an individual in the domain.

2.3 Laconicity and Construct Redundancy

A model M is called *laconic* w.r.t. an abstraction A if the interpretation mapping from M to A is *injective*, i.e., iff every entity in the abstraction A is represented by *at most* one (although perhaps none) entity in the model M. An example of an injective interpretation mapping is depicted in Fig. 2c. The notion of *laconicity* in the level of individual specifications is related to the notion of *construct redundancy* in the language level in [6]: *"construct redundancy occurs when more than one grammatical construct can be used to represent the same ontological construct"*.

Once again, despite being related, laconicity and construct redundancy are two different (even opposite) notions. On the one hand, construct redundancy does not entail non-laconicity. For example, a language can have two different constructs to represent the same concept. However, in every situation the construct is used in

particular specifications, it only represents a single domain element. On the other hand, the lack of construct redundancy in a language does not prevent the creation of non-laconic specifications in that language. For example, the *arrow/labeled box* language for representing natural numbers in Sect. 2.2 is laconic, i.e., for each domain type there is at most one construct type in the language. However, we can still produce using this simple language a specification in which, for example, the same natural number (e.g., 3) is represented by more than one labeled box.

Non-laconicity can also be manifested at the language level. We say that a language is non-laconic if it has a non-laconic modeling construct, i.e., a construct that when used in a specification of a model causes an entity of this model to be represented more than once. Non-laconicity at the language level is a special case of construct redundancy that does entail non-laconicity at the level of individual diagrams.

In [6], the authors claim that construct redundancy *"adds unnecessarily to the complexity of the modeling language"* and that *"unless users have in-depth knowledge of the grammar, they may be confused by the redundant construct. They might assume, for example, that the construct somehow stands for some other type of phenomenon"*. Therefore, construct redundancy can also be considered to undermine representation clarity. In summary, a modeling language should not contain construct redundancy, and elements in the represented domain should be represented by at most one instance of the language modeling constructs.

2.4 Completeness

A model M is called *complete* w.r.t. an abstraction A if an interpretation mapping from M to A is *surjective*. An interpretation mapping from M to A is surjective iff the corresponding representation mapping from A to M is total, i.e., iff every entity in an abstraction A (instance of a domain conceptualization) is represented by *at least* one (although perhaps many) entity in the model M. An example of a *surjective* interpretation mapping is depicted in Fig. 2d.

The notion of completeness at the level of individual specifications is related to the notion of *ontological expressiveness* and, more specifically, *completeness* at the language level, which is perhaps the most important property that should hold for a representation system. A modeling language is said to be *complete* if every concept in a domain conceptualization is covered by at least one modeling construct of the language. Language incompleteness (also termed *Construct Deficit*) entails lack of expressivity, i.e., that there are phenomena in the considered domain (according to a conceptualization) that cannot be represented by the language. Alternatively, users of the language can choose to overload an existing construct, thus, undermining clarity.

An incomplete modeling language is bound to produce incomplete specifications unless some existing construct is overloaded. However, the converse is not true, i.e., a complete language can still be used to produce incomplete specifications.

Once more, we refer to the arrow/labeled box language of previous sections. This language is complete, i.e., for each domain type there is at least one construct type in the language. However, we can still produce using this language a specification in which, for example, a relation instance is missing (e.g., the less-than relation between the boxes representing numbers 2 and 3).

3 Conceptualization, Ontology, and Meta-Model

Let us now return our attention to Fig. 1. A modeling language can be seen as delimiting all possible specifications[1] which can be constructed using that language, i.e., all grammatically valid specifications of that language. Likewise, a conceptualization can be seen as delimiting all possible domain abstractions (representing state of affairs) which are admissible in that domain [8]. Therefore, for example, in a conceptualization of the domain of genealogy, there cannot be a domain abstraction in which a person is his own biological parent, because such a state of affairs cannot happen in reality. Accordingly, we can say that a modeling language that is truthful to this domain is one that has as valid (i.e., grammatically correct) specifications only those that represent state of affairs deemed admissible by a conceptualization of that domain. In the sequel, we review a formalization of this idea presented at [2], which is an extension of the original idea proposed in [8]. This formalization compares conceptualizations as *intentional structures* and meta-models as represented by logical theories. Thus, in the sequel, we make use of the terms *possible world*, *domain of quantification*, *relation,* and *interpretation function* in their traditional established sense in model logics [9].

Let us first define a *conceptualization* C as an intensional structure $\langle W,D,\Re\rangle$ such that W is a (non-empty) set of possible worlds, D is the domain of individuals and $\Re$ is the set of relations (concepts) that are considered in C. The elements $\rho \in \Re$ are intensional (or conceptual) relations with signatures such as $\rho^n{:}W \rightarrow \wp(D^n)$, such that n is the arity of ρ, and so that each relation is a function from possible worlds to sets of n-tuples of individuals in the domain. For instance, we can have ρ accounting for the meaning of the natural kind Apple. In this case, the meaning of Apple is captured by the intentional function ρ, which refers to all instances of Apple in every possible world. For every world $w \in W$, according to C we have an *intended world structure* S_wC as a structure $\langle D, R_wC\rangle$ such that $R_wC = \{\rho(w) \mid \rho \in \Re\}$. More informally, we can say that every intended world structure S_wC is the characterization of some state of affairs in world w deemed admissible by conceptualization C. From a complementary perspective, C defines

[1] We have so far used the term *model* instead of specification since it is the most common term in conceptual modeling. In this session, exclusively, we adopt the latter in order to avoid confusion with the term (logical) model as used in logics and Tarskian semantics. A specification here is a syntactic notion; a logical model is a semantic one.

all the admissible state of affairs in that domain, which are represented by the set $S_c = \{S_wC \mid w \in W\}$.

Let us consider now a language L with a vocabulary V that contains terms to represent every concept in C. A logical model for L can be defined as a structure $\langle S,I \rangle$: S is the structure $\langle D,R \rangle$, where D is the domain of individuals and R is a set of extensional relations; $I:V \rightarrow D \cup R$ is an interpretation function assigning elements of D to constant symbols in V, and elements of R to predicate symbols of V. A model, such as this one, fixes a particular extensional interpretation of language L. Analogously, we can define an intensional interpretation by means of the structure $\langle C,\Im \rangle$, where $C = \langle W,D,\Re \rangle$ is a conceptualization and $\Im:V \rightarrow D \cup \Re$ is an intensional interpretation function which assigns elements of D to constant symbols in V, and elements of $\Re$ to predicate symbols in V. In [8], this intensional structure is named the *ontological commitment* of language L to a conceptualization C. We therefore consider this intensional relation as a formal characterization of the *represented by* relation depicted in Fig. 1, or simply a formal characterization of the Real-World Semantics of L [10].

Given a logical language L with vocabulary V, an *ontological commitment* $K = \langle C,\Im \rangle$, a model $\langle S,I \rangle$ of L is said to be compatible with K if: (i) $S \in S_c$; (ii) for each constant c, $I(c) = \Im(c)$; (iii) there exists a world w such that for every predicate symbol p, I maps such a predicate to an admissible extension of $\Im(p)$, i.e., there is a conceptual relation ρ such that $\Im(p) = \rho$ and $\rho(w) = I(p)$. The set $I_k(L)$ of all models of L that are compatible with K is named the set of *intended models* of L according to K.

Finally, given a specification X in a specification language L, we define as the logical rendering of X, the logical theory T that is the first-order logic description of that specification [11].

In order to exemplify these ideas let us take the example of a very simple conceptualization C such that $W = \{w,w'\}$, $D = \{Gordon, Andy, Stewart\}$ and $\Re = \{person, father\}$. Moreover, we have that person(w) = {Gordon, Andy, Stewart}, father(w) = {Gordon}, person(w') = {Gordon, Andy, Stewart} and father(w') = {Gordon, Stewart}. This conceptualization accepts two possible state of affairs, which are represented by the world structures S_wC = {{Gordon, Andy, Stewart}, {{Gordon, Andy, Stewart}, {Gordon}}} and $S_{w'}C$ = {{Gordon, Andy, Stewart}, {{Gordon, Andy, Stewart}, {Gordon, Stewart}}}. Now, let us take a language L whose vocabulary is comprised of the terms Person and Father with an underlying meta-model that poses no restrictions on the use of these primitives. In other words, the meta-model of L has the following logical rendering (T_1): $\{\exists x\ Person(x), \exists x\ Father(x)\}$. In this case, we can clearly produce a logical model of L (i.e., an interpretation that validates the logical rendering of L) but that is not an intended world structure of C. For instance, the model D' = {Gordon, Andy, Stewart}, person = {Gordon, Andy}, father = {Stewart}, and I(Person) = person and I(Father) = father. This means that we can produce a specification using L which has a model that is not an *intended model* according to C.

Now, let us update the meta-model of language L by adding one specific constraint and, hence, producing the meta-model (T_2): $\{\exists x\ Person(x), \exists x\ Father(x),$

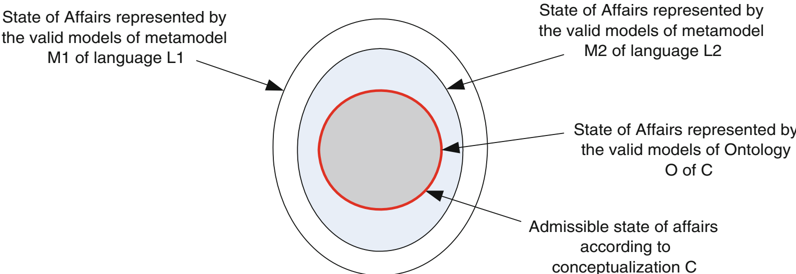


Fig. 3 Measuring the degree of *domain appropriateness* of modeling languages via an ontology of a conceptualization of that domain

$\forall$x Father(x) $\rightarrow$ Person(x)}. Contrary to L, the resulting language L' with the amended meta-model T_2 has the desirable property that all *its valid specifications have logical models that are intended world structures of C.*

A domain conceptualization C can be understood as describing the set of all possible state of affairs, which are considered admissible in a given universe of discourse U. Let V be a vocabulary whose terms directly correspond to the intensional relations in C. Now, let X be a *conceptual specification* (i.e., a concrete representation) of universe of discourse U in terms of the vocabulary V and let T_X be a logical rendering of X, such that its formal constraints restricts the possible interpretations of the members of V. We call X (and T_X) an *Ontology* of U according to C iff the logical models of T_X describe all and only state of affairs which are admitted by C. This use of the term ontology is strongly related to a definition of Ontology put forth by the philosopher W.V.O. Quine, i.e., ontology as *a theory concerning the kinds of entities and specifically the kinds of abstract entities that are to be admitted to a language system* [2].

With an explicit representation of a conceptualization in terms of a suitable ontology, one can measure the truthfulness (or *domain appropriateness*) of a language L to domain D, by observing the difference between the set of logical models of the (logical rendering of) meta-model M of L and the set of logical models of the (logical rendering of) ontology O of D (see Fig. 3). In the ideal case, these two specifications are isomorphic and, hence, share the same set of logical models. Therefore, not only every entity in conceptualization C must have a representation in the meta-model M of language L, but these representations must obey the same axiomatization.

According to the language evaluation framework and the formal characterization of the relation between ontology and language vocabulary defined here, we can provide the following characterization for an ideal language to represent phenomena in a given domain according to a given reference ontology:

> A language is ideal to represent phenomena in a given domain if the metamodel of this language is isomorphic to the reference ontology of that domain and the language only has as valid specifications those whose logical models are exactly the logical models of that reference ontology.

The traditional account of ontological analysis of languages in the literature is articulated in terms of isomorphism between language and ontology (such as in [5, 6]). The above definition relates this traditional account to a formal definition of ontology as a *formal and explicit specification of a conceptualization* [8]. There is one direct manner in which incompleteness (and hence, lack of isomorphism) can impact the quality of language L, namely, when the meta-model M of L does not contain constructs to fully characterize a state of affairs, and therefore to produce the axiomatization necessary to exclude unintended logical models of the conceptualization at hand. To give one example, in the genealogical domain, without a gender differentiation for people, one cannot produce an axiomatization which excludes models in which people have two individuals of the same gender as their biological parents. Additionally, as exemplified in Sect. 2, without the proper formal constraints in its meta-model, even lucid, sound, laconic, and complete representation systems can be used to produce specifications lacking these desirable characteristics.

4 Successful Cases of General Conceptual Modeling Languages Evaluation and Re-design Using the Proposed Approach

The definition of an *ideal conceptual modeling language* given in Sect. 3 provides clear guidelines for the design of the ontological meta-models of these languages. Given a reference ontology, the meta-model at hand should be isomorphic to the ontology of the domain. Moreover, it should include formal constraints such that the language would only accept as grammatically correct models those that represent state of affairs deemed admissible by the ontology at hand. Finally, this ontological meta-model should be further enriched with additional formal constraints that guarantee lucidity, laconicity, completeness, and soundness for all individual diagrams that can be produced using that language.

These guidelines as advocated here are agnostic regarding the type of language which is being evaluated or (re)designed, meaning, this framework can be employed both for the case of general conceptual modeling languages (e.g., UML, ER, ORM, BPMN) and the case of *Domain-Specific Languages*. The type of language considered, however, is directly related to the type of ontology which can be used as a reference model. In the case of general (hence, domain-independent) conceptual modeling languages, the required reference ontology is a *Foundational Ontology*, i.e., a domain-independent system of categories and their ties which can be used to articulate models of different material domains in reality. In contrast, for the case of domain-specific languages, the required reference ontology is a *Domain Ontology* [2].

Two examples of Foundational Ontologies which have been successfully used for evaluating general conceptual modeling languages over the years are BWW [6, 12]

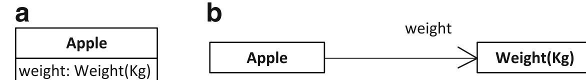


Fig. 4 Redundant representation of Attribute Functions in UML

and UFO [10, 13]: BWW has been used to analyze languages such as ARIS [14] and OWL [15], among others. The application of BWW for this purpose has traditionally been carried out by employing the original proposal put forth in [5]. However, none of these analyses have considered the axiomatization of the ontology (in terms of its admissible models) or the formal constraints incorporated in the language meta-models.

For a number of years, we have been analyzing conceptual modeling languages (including enterprise modeling languages), standards, environments, and domain ontologies, by employing the method described above and the foundational ontology UFO (Unified Foundational Ontology) as a reference model. The analyzed modeling languages include: UML [10, 16–19], Archimate [20], RM-ODP [21], TROPOS/i* and AORML [22, 23], ARIS [24], and BPMN [25]. Despite the successful application of UFO in all these cases, it is important to highlight that the method discussed here could, in principle, be applied by taking different foundational ontologies as reference models. In fact, preliminary results on our ontological analysis of UML have been carried out with the foundational ontology GFO [26].

One significant case of ontological analysis using the framework discussed here and which deserves special attention is the case of UML. When considered as a Conceptual Modeling language, UML alone includes cases of all the anomalies discussed above. An example of *Construct Excess* in UML relates to the *Interface* Construct. As discussed in [10], being merely a design and implementation construct, there is no category in the reference ontology that serves as the ontological interpretation for a UML interface. Moreover, construct excess in the language level will cause unsoundness in all diagrams in which the exceeding construct is employed. UML also presents at least one case of *Non-Lucidity*, namely, in the *Association Class* construct. More than a case of *Construct Overload*, in each and every occasions this construct is used, it will stand for two ontological entities simultaneously, namely, a *Relator Universal* (e.g., Marriage, Enrollment, Employment) whose instances are individual relators (e.g., the Marriage of Mary and John, the Enrollment of Zoe to UFES) and a *Factual Universal* whose instance are tuples (e.g., pairs such as <John,Mary> and <Mary,John>) [10].

Another case of Construct Overload is the construct of *navigable ends* in UML which can be used to represent both *Relational Image Functions* (also known as mappings) and *Attribute Functions* [10]. Actually, also related to Attribute Functions, we have a case of *Construct Redundancy*, since attributes can be represented both by: the traditional textual representation of attributes spatially contained in the Class representation (Fig. 4a) and navigable ends (Fig. 4b).

Finally, cases of *Construct Deficit* (*Ontological Incompleteness*) in UML abound. For example, there are several different sorts of *object types* and *part-whole relations* in the conceptualizations proposed in [17] and [18], respectively, which are

not directly represented by any construct of the language. In both cases, the distinct concepts present in the conceptualization are overloaded by the language constructs of *class* and *aggregation/composition*, respectively. To cite just one more example, in [19], we have shown that the concept of Mode (an ontological counterpart to the ER notion of Weak Entity) also finds no direct representation in the language.

In [10], a philosophically and cognitively motivated foundational ontology (later identified as the type-fragment of UFO-A) has been used to redesign a complete version of the class diagrams fragment of the UML 2.0 meta-model giving rise to a well-founded version of UML for structural conceptual modeling and domain ontology representation. This ontology representation language (later dubbed *OntoUML*) has been successfully employed to create domain ontologies in several different industrial case studies in domains such as Telecommunications [27] and Energy (Petroleum and Gas) [28]. Moreover, it has been used to support meaning negotiation and semantic interoperability in the integration of ECG standards [29]. Furthermore, a version of this language has been employed over the past years by a department of the DoD in a significant number of successful applications in real-world engineering settings.[2]

Aside from the discussed cases of Construct Overload, Excess, Deficit, Redundancy, and Non-Lucidity at the language level, the weakly constrained original UML meta-model accepts a number of instances which represent ontologically inadmissible ontological structures. All these problems have been addressed in [10] as well as in follow-up publications such as [30, 33]. As a result, the revised UML meta-model (i.e., the OntoUML meta-model) is isomorphic to the ontological distinctions comprising the underlying foundational ontology and includes as formal constraints representations of the ontological constraints. Due to this strategy, these ontological distinctions and constraints could be directly implemented using meta-modeling architectures such as the OMG's MOF (Meta Object Facility).[3] In this line, [31] reports on an implementation of an OntoUML graphical editor, which applies such an approach for assisting the user in creating ontologically correct models.

5 Ontological Meta-Properties, Model Simulation, and Domain-Specific Visual Languages

Aside from being an extensive and successful evaluation case of the framework discussed here, OntoUML constitutes in itself a contribution to the application of this framework in the level of material domains and, thus, in the evaluation and design of *Domain-Specific Languages*. The most obvious reason is the following:

[2] http://www.omgwiki.org/architecture-ecosystem/lib/exe/fetch.php?media=dmg_for_enterprise_ldm_v2_3.pdf

[3] http://www.omg.org/mof/

the only grammatically correct models in OntoUML are ontologically consistent models; OntoUML can be used to represent structural conceptual models, in general, and domain ontologies in particular; thus, the domain ontologies constructed in OntoUML will be consistent with the axiomatization of the underlying foundational ontology.

However, there are two other reasons for why this language plays an important role in the design of domain ontologies that will, in turn, be used for the evaluation and design of domain-specific visual modeling languages. Firstly, aside from its model-theoretical semantics defined in [10], OntoUML has an operational semantics defined as a mapping from this language to the lightweight formal language Alloy [32]. Due to this mapping, we have configured the Alloy Analyzer tool[4] so that it can be used for supporting the modeler in assessing the gap between the *intended models* (in the sense of Sect. 3) and the *possible models* of the domain ontology at hand (and, hence, of a possible meta-model isomorphic to it). This topic is discussed and illustrated in Sects. 5.2 and 5.3. Secondly, contrary to the merely formal (algebraic) structures employed in [3], the domain ontologies represented in OntoUML capture a number of subtle ontological meta-properties that are used to further qualify the ontological status of the domain concepts. In Sect. 5.4, we illustrate how the ontological meta-properties can be systematically exploited to improve the system of concrete syntax of domain-specific visual modeling languages.

5.1 Ontological Meta-Properties

Due to space limitations, we concentrate here on a fragment of OntoUML, with a focus on distinctions among *Object Types* and *Part-Whole* relations spawned by variations in ontological meta-properties.

A fundamental modal meta-property used to distinguish among categories of *Object Types* is *Rigidity* (and the associated notion of *Anti-Rigidity*). Formally, we have that [17]: a type T is rigid iff every instance of T is necessarily an instance of T (in the modal sense). In contrast, a type T' is anti-rigid iff for every instance x of T' there is a possible situation in which x is not an instance of T'. In other words, an instance of a rigid type T cannot cease to instantiate it without ceasing to exist. Contrariwise, instances of T' only instantiate it contingently and, hence, can move in and out of the extension of T' without altering their identity. A stereotypical example that illustrates this distinction in most conceptualizations is marked by the types Person and Student: instances of Person are necessarily so (thus, Person is a rigid type); in opposition, instances of Student are merely contingently so (thus, Student is an anti-rigid type).

Object types that are rigid are named *Kinds* and *Subkinds* [17]. These types define a stable backbone, i.e., a taxonomy of rigid types instantiated by a given individual (the kind being the unique top-most rigid type instantiated by an individual).

[4] http://www.alloy.mit.edu/

Within the category of anti-rigid object types, we have a further distinction between *Phases* and *Roles* [17]. Both Phases and Roles are specializations of rigid types (Kinds/subKinds). However, they are differentiated w.r.t. their *specialization conditions*. For the case of Phases, the specialization condition is always an intrinsic one. For instance, a Child is a Person whose age is within a certain range. In contrast, the specialization condition for Roles is a relational one. For instance, a Student is a Person who is enrolled in an Educational Institution.

Again, a modal meta-property used to distinguish among the categories of Part-Whole relations is *Existential Dependence* [18]. We have that an entity x is existentially dependent on another entity y iff in every situation that x exists then y must exist. Associated with Existential Dependence we have the notion of Generic Dependence. We have that an entity x is generically dependent on a type Y iff in every situation where x exists an instance of Y must exist. These notions are used in UFO (among many other things) to distinguish between part-whole relations that imply existential dependence and those that only imply generic dependence. A part-whole relation which implies only generic dependence from the part to the whole is named *parthood with mandatory wholes* [18]. In contrast, a part-whole relation that implies existential dependence from the part to the whole is termed *inseparable parthood* [18]. A stereotypical example that illustrates this distinction in most conceptualizations is marked by the types of the relation between a Heart and a Person, on one side, and between a Brain and a *particular* Person, on the other: while a Heart needs to be part of an instance of Person (which does not have to be the same in every possible situation), a Brain needs to be part of a specific Person in all situations in which it exists.

Another remark regarding part-whole relations worth mentioning here is the following: contrary to purely formal mereological relations, part-whole relations which appear in conceptual models and material domain ontologies are non-transitive, i.e., they are transitive in certain situations and intransitive in others [33]. As illustrated in Sect. 5.4, assessing the correct value of this additional meta-property of part-whole relations has an important influence on the design of their concrete visual representations.

Finally, part-whole relations can be distinguished according to a meta-property named *shareability*. This meta-property wrongly defined in the original UML specification has been refined in [10] with the following definition: (a) a (whole) type X is characterized by an exclusive (non-shareable) parthood relation with a (part) type Y iff every instance of X must have exactly one instance of Y as part; (b) a type X is characterized by a shareable parthood relation with a type Y iff instances of X can have more than one instance of Y as part.

5.2 Contrasting Possible and Intended Models with Visual Model Simulation

A modeling language such as OntoUML, incorporating the ontological constraints of a foundational theory prevents the representation of ontologically non-admissible

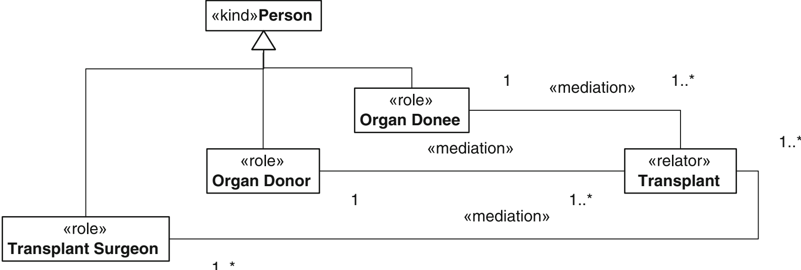


Fig. 5 A fragment of a fictitious ontology in which unintended instances are admitted

states of affair in domain ontologies represented in that language. However, it cannot guarantee that only *intended states of affairs* are represented by the domain model at hand. This is because the admissibility of domain-specific states of affair is a matter of factual knowledge (regarding the world being the way it happens to be), not a matter of consistent possibility.

To illustrate this point, suppose a medical domain ontology representing the procedure of a transplant. In this case, we have domain concepts such as Person, Transplant Surgeon, Transplant, Transplanted Organ, Organ Donor, and Organ Donee. The (obviously incomplete) model of Fig. 5, which models aspects of this situation, does not violate any ontological rule. It would be the case, for example, had we placed Organ Donor as a super-type of Person, or represented the possibility of a Transplant without participants. These two cases can be easily detected and proscribed by an editor such as the one just proposed in [31]. However, there are still unintended states of affairs (according to a conceptualization assumed here) that are represented by valid instances of this model. One example is a state of affairs in which the Donor, the Donee and the Transplant Surgeon are one and the same Person. Please note that this state of affairs is only considered inadmissible due to domain-specific knowledge of social and natural laws. Consequently, it cannot be ruled out a priori by a domain independent system of ontological categories.

Guaranteeing the exclusion of unintended states of affairs without a computational support is a practically impossible task for any relevant domain. In particular, given that many fundamental ontological distinctions are modal in nature, in order to validate a model, one would have to take into consideration the possible valid instances of that model in all possible worlds.

In [34], we have proposed an approach for OntoUML which offers a contribution to this problem by supporting conceptual model validation via visual simulation. On the one hand, it aims at proving the satisfiability of a given ontology by presenting a valid instance (logical model) of that ontology. On the other hand, it attempts to exhaustively generate instances of the ontology in a branching-time temporal structure, thus, serving as a visual simulator for the possible dynamics of entity creation, classification, association, and destruction. The snapshots in this world structure confront a modeler with states of affairs that are deemed admissible by

the ontology's current axiomatization. This enables modelers to detect unintended states of affairs and to take the proper measures to rectify the model. The assumption is that the example world structures support a modeler in this validation process, especially since it reveals how states of affairs change in time and how they may eventually evolve in counterfactual scenarios.

After running simulations of the model of Fig. 5, the model engineer would be presented with the consequences of her specification. When faced with a situation in which the Donor, Donee and Surgeon roles are played by the same person, she could realize that the ontology at hand has been *under-constrained* and then include a constraint in the model to exclude this unintended situation. Now, suppose the situation in which the modeler tries to rectify this model by declaring the types Transplant Surgeon, Organ Donor, and Organ Donee as mutually disjoint. In a follow-up execution of simulating this ontology, she would then realize that it is not possible, for example, for an Organ Donor to receive an organ in a different transplant, and for a Transplant Surgeon to be either an Organ Donor or an Organ Donee in different transplants. When facing this new simulation results, the modeler could realize that now the ontology has been *over-constrained,* after all there is no problem in having the same person as Organ Donor and Donee, or as Surgeon and Donor (Donee), they only cannot play more than one of these roles in the same transplant! In summary, the idea is that in this multi-step interaction with the model simulator, the modeler can keep refining the domain constraints to increasingly approximate the possible model instances of the ontology to those that represent admissible states of affairs according to the underlying conceptualization. In addition, in line with [35], we advocate that *"simulation helps catch errors of overconstraint, by reporting, contrary to the user's intent, that no instance exists within the finite bounds of a given "scope," or errors of underconstraint, 'by showing instances that are acceptable to the specification but which violate an intended property".*

5.3 From a Domain Ontology to the Design of a Domain-Specific Conceptual Modeling Language Meta-Model

In Fig. 6 below, we have a small ontology fragment in the domain of organizations represented using OntoUML. In the underlying conceptualization, *Employee* is a role played by people associated with one *Department*. People (instances of *Person*) are either instances of *Man* or of *Woman*, i.e., *Person* is an abstract type in the sense of object-oriented modeling, meaning there is no one who is a Person without being either a *Man* or a *Woman*. Every *Employee* is *part of* exactly one *Department* (represented by the non-shareable association end). However, since this is merely a generic dependence relation, employees can in principle change to different departments (even in different branches) in their life cycle in the organization. An

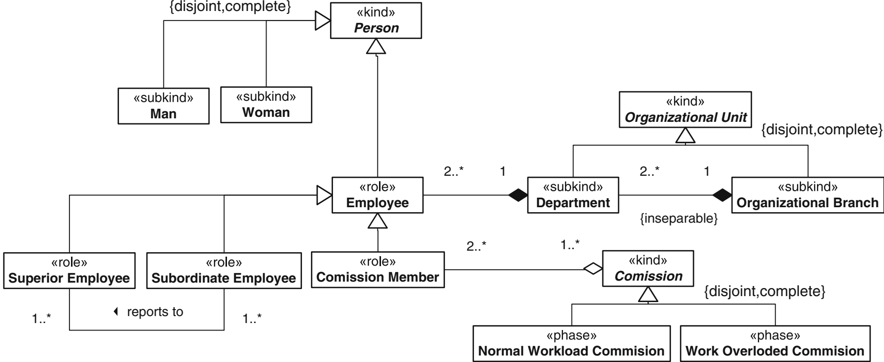


Fig. 6 A domain ontology for an organizational domain used to create a meta-model for a domain-specific language

Employee is subordinated to at least one other employee who is his/her superior. In other words, the types Subordinate Employee and Superior are roles played by employees in the scope of hierarchical relations. As roles, these types are only contingently instantiated by their instances and the relational specialization condition here is represented by the *reports-to* relation. In other words, an instance of *Subordinate Employee* can cease to be one, and for her to instantiate this role, there must exist another *Employee* instantiating the *Superior Employee* role. Moreover, the same instance of Employee can simultaneously instantiate both roles in the scope of different hierarchical relations (i.e., being the *Superior* of *Employee* A and subordinated to Employee B).

Every *Department* is *part of* exactly one *Organizational Branch*. Here, again, we have not only the case of a non-shareable parthood relation but also one which implies existential dependency from the part to the whole (represented by the {*inseparable*} tag value), i.e., the Sales Department of one Organizational Branch can only exist as part of that Branch. The relation between *Employee* and *Department*, on the one hand, and *Department* and *Organizational Branch*, on the other, matches one of the cases of transitive parthood identified in [33]. For this reason, we have that: if an *Employee* E is part of a *Department* D and D is part of the *Organizational Branch* O, then E is part of O.

Commissions are collectives that have particular *Employees* as members (accordingly termed *Commission Members*). Commissions can be in two different phases depending on the value of one of its intrinsic property (its amount of committed work). Thus, a *Work-Overloaded Commission* is a *Commission* such that its amount of committed work surpasses a certain threshold. Finally, a *Normal Workload Commission* is the complement of *Commission* w.r.t. *Work-Overloaded Commission*, i.e., its instances are all instances of the former which are not instances of the latter.

Besides the representation of all relevant domain types, relations, and properties of the underlying conceptualization, a domain ontology (and the meta-model derived from it) must include a body of formal constraints. This axiomatization

must restrict the states of affairs represented by valid models of this ontology/meta-model to those which represent intended states of affairs according to the underlying conceptualization. As discussed in Sect. 3, the quality of this ontology depends on the distance between these two sets of states of affairs. Moreover, as discussed in the previous section, for the case of OntoUML we can count on an approach for model validation via visual simulation.

Figures 7a, b present results of a visual simulation of the ontology in Fig. 6. By looking at these automatically generated models, the modeler can realize the lack of the missing partial order constraints that should be defined for the relation *reports to*. In Fig. 7a, an Employee plays the roles of Superior Employee and Subordinate Employee in the same relation, i.e., the employee reports to herself. In Fig. 7b, we can notice that Man0 is subordinate to both Woman2 and Woman0, who are then subordinate to Woman1, who then is subordinate to Man0. In other words, the model admits cycles in the *reports to* relation. Still on the model of Fig. 7b, one can notice that although Man0 is subordinate to Woman2 and Woman0, both who are subordinate to Woman1. However, Man0 is not subordinate to Woman1, i.e., the *reports to* relation is not considered to be transitive. In Fig. 7c, one can notice the possibility of an employee who falls outside the hierarchical structure of the organization, i.e., who is neither subordinate nor superior to anyone. This is due to a missing {*complete*} constraint in the generalization set from Employee to Superior Employee and Subordinate Employee. Finally, in the model of Fig. 7d, one can notice the situation in which an employee (Man1) reports to a superior (Man0) of a different department. As previously discussed, these models cannot be deemed undesirable due to general ontological rules but only due to domain-specific rules. If we assume that in the conceptualization underlying the ontology of Fig. 6 these are all unintended models, then when facing them as possible ones, the modeler can improve the ontology (and corresponding Domain-Specific Language Meta-model) at hand by including the constraints required for their exclusion.

5.4 From a Domain Ontology to the Design of Domain-Specific Visual Modeling Language Concrete Syntax

In this section, we discuss the impact that the reference ontology also has in the design of a concrete syntax for a visual conceptual modeling language. In order to do that, we base our discussion in the framework for analysis and design of visual languages put forth by Daniel Moody in [36, 37].

The most direct influence that an ontology has on the visual notation of a language regards the quality that Moody terms *Semiotic Clarity*. By discussing Semiotic Clarity, Moody conducts an analysis similar to the one put forth here, but now relating ontology and visual concrete syntax. He draws from Nelson Goodman's *Theory of Symbols* when advocating *for a notation to satisfy the requirements of a notational system, there should be a one-to-one correspondence between symbols and their referent concepts* [38]. Figure 8 below (from [37]) summarizes

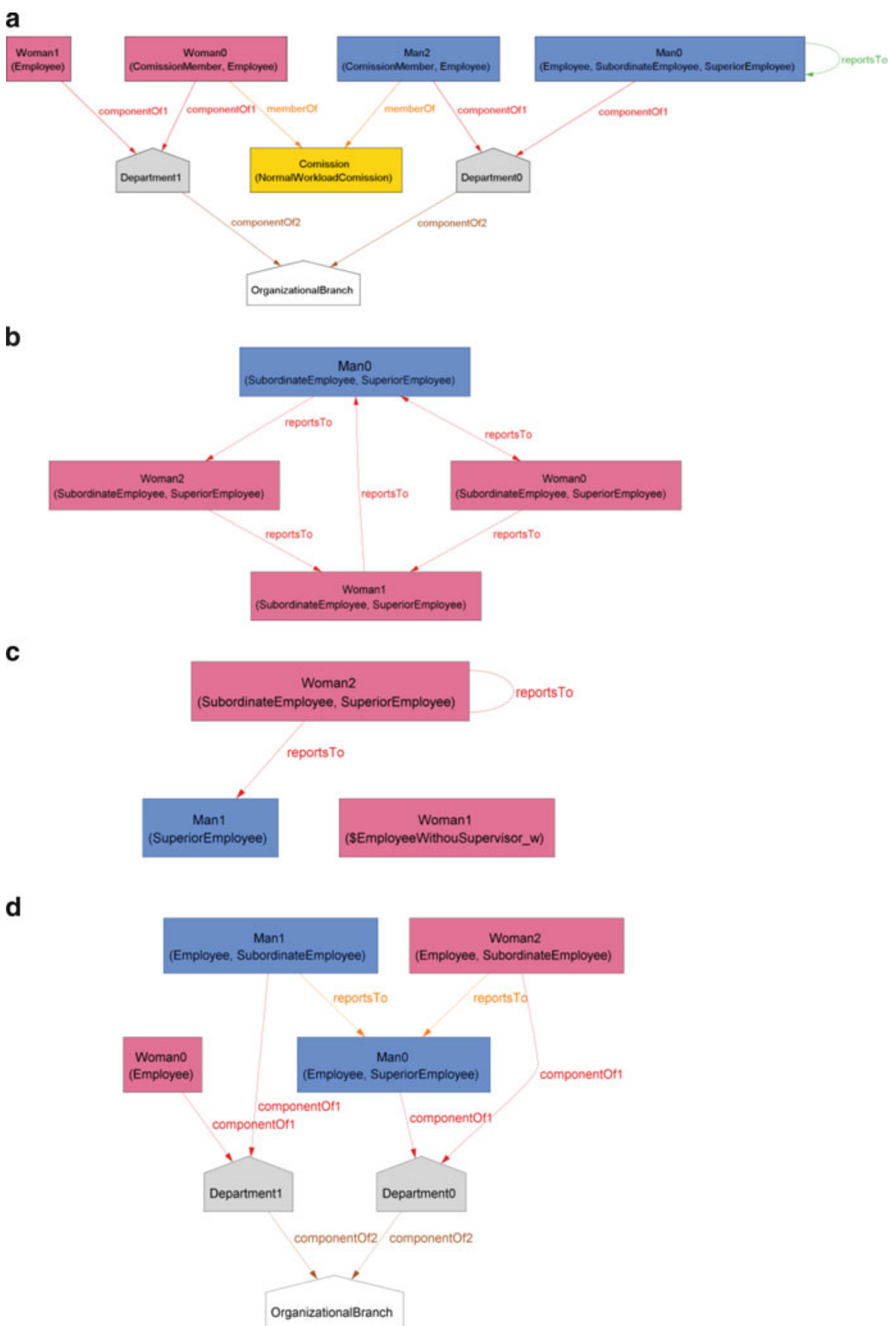


Fig. 7 Examples of possible but unintended instances of the ontology in Fig. 6: (**a**) the employee that is his own superior; (**b**) cyclic hierarchies; (**c**) employee outside the organizational structure; (**d**) employee with a superior in a different department

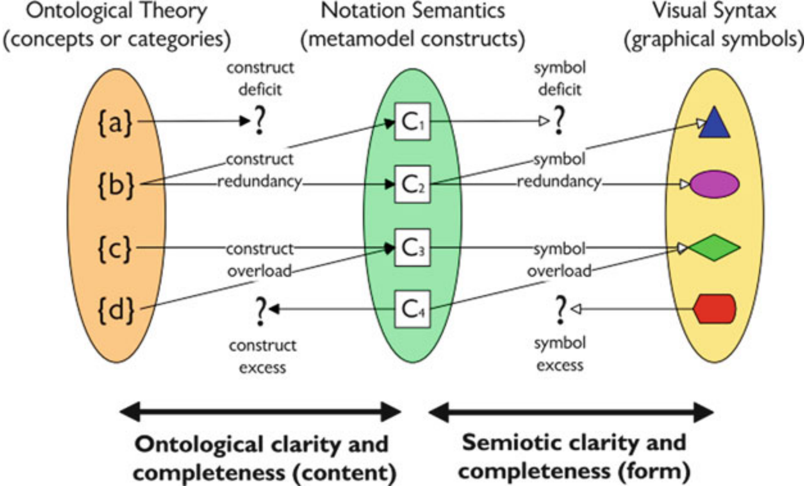


Fig. 8 From ontological concepts to language primitives to visual syntax (from [37]). Source © Springer-Verlag 2009, Moody, D.L., van Hillegersberg, J.: Evaluating the visual syntax of UML: an analysis of the cognitive effectiveness of the UML family of diagrams. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 16–34. Springer, Heidelberg (2009)

this correspondence between the Ontological Meta-model of the language and the underlying domain ontology, on the one hand, and between the Ontological Meta-model of the language and the description of the concrete syntax, on the other.

In classifying the anomalies that take places when the isomorphism between the latter pair of models is broken, Moody builds explicitly on the vocabulary used in the literature of ontological analysis: (a) *Symbol redundancy* exists when multiple symbols are used to represent the same semantic construct; (b) *Symbol overload* exists when the same graphical symbol is used to represent different semantic constructs; (c) *Symbol excess* exists when graphical symbols are used that do not represent any semantic construct; (d) *Symbol deficit* exists when semantic constructs are not represented by any graphical symbol.

The problems caused by these anomalies are, as explained by Moody, also analogous to those founded when the isomorphism between ontology and abstract syntax is broken: symbol redundancy places a burden of choice on the language user to decide which symbol to use and an additional load on the reader to remember multiple representations of the same construct; Symbol overload leads to ambiguity and the potential for misinterpretation [38]; symbol excess unnecessarily increases graphic complexity, which has been found to reduce understanding of notations [39]. Moreover, if symbol deficit exists, the visual notation is said to be *semiotically incomplete*. If any of the other three anomalies exist, the notation is *semiotically unclear*.

There is an obvious connection here with what we have been discussing so far: the suitability of a visual notation is evaluated w.r.t. a system of modeling primitives, which in turn is evaluated w.r.t. to a domain ontology. Hence, the

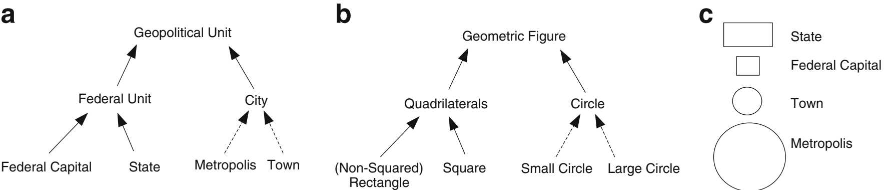


Fig. 9 (**a**) A fragment of taxonomy for a geopolitical domain; (**b**) a taxonomy of geometric objects isomorphic to the structure in (**a**); (**c**) a system of visual symbols from (**b**) to represent the domain concepts in (**a**)

quality of a system of visual syntax w.r.t. semiotic clarity indirectly but essentially depends on the characteristics of the underlying domain represented in that domain ontology. One aspect, however, which is not evident in Moody's model above, is the following. As previously discussed, the graphical symbols which form the system of concrete syntax often fall naturally into a hierarchical typing which informs about the semantics of what is being represented. An analogous statement can be made regarding certain relations between graphical symbols (e.g., spatial relations) that can be systematically mapped onto semantic relations with equivalent logical properties. This feature of graphical symbol systems and relations is illustrated by the examples in the sequel.

We start with a first example illustrated in Fig. 9. As one can notice, the models of Fig. 9a, b are isomorphic. The different concrete kinds of entities in the model (Federal Capital, State and City) are represented by different kinds of geometrical objects (Non-Squared Rectangle, Square and Circle). In particular, the taxonomic structure of Geopolitical Units is isomorphic to the one of Geometric Figures. For this reason, in a visual query one can immediately notice that Federal Capital is more similar to a State than to a City and probably share a common super-type with the former. Notice that had we produced a different taxonomic structure on the model of Fig. 9a, then a different choice of representing graphical symbols would have been made possibly creating undesired implicatures for the model reader. Given the difficulties experienced by modelers in the design of domain taxonomic structures [17], this illustrates the importance of having a well-designed ontology for the design of semiotic clarity in the system of visual syntax.

There is another point worth mentioning about this example. As discussed in the previous section, a phase represents a type that is instantiated by an individual only contingently (in the modal sense) and changes of phases are motivated by changes in intrinsic properties. In this example, the same city can be considered a town in a world w and a metropolis in w' while still maintaining its cross-world identity. Likewise, in Fig. 9b, the size property of a geometric figure is considered one of its contingent intrinsic properties. Thus, a particular circular form is assumed to be able to change its size while maintaining a continuous visual percept. Furthermore, the intrinsic property *population size* that motivates the phase changes of cities is associated with a linearly ordered dimension. For this reason, we have decided

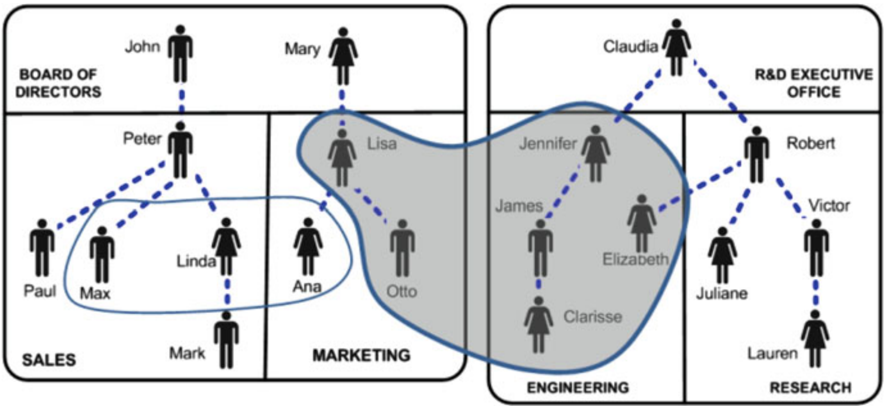


Fig. 10 A model in the domain-specific language to represent organizational structures

to associate the intrinsic property of Circles that represent this phase variation by employing also a linearly ordered dimension (size).

As a second example, we refer to the model of Fig. 6. In Fig. 10 below, we present a model in a domain-specific visual language designed to represent valid instances of the ontology of Fig. 6. There are a number of aspects in the concrete syntax of this visual language that have been designed by systematically considering logical and ontological meta-properties of the domain concepts in Fig. 6. Firstly, this system of concrete syntax possesses Semiotic Clarity, i.e., there is a one-to-one correspondence between its categories and the domain types and relations represented in Fig. 6. Secondly, the mapping between the domain elements and the elements in the visual notation takes full account of the ontological categories and meta-properties of the former. In the sequel, we elaborate on this second point.

Kinds and Subkinds: In the model of Fig. 6, we have three kinds of elements, namely, Person, Organizational Units and Commission. Since Person is an abstract type, we have that all instances of this type in the domain are instances of either Man or Woman. Likewise, all instances of Organizational Units are either Organizational Branches or Departments. In summary, all the concrete substantial entities in this domain are either instances of Man, Woman, Departments, Organizational Branches, or Commissions.

As discussed in [40], shapes defined by closed contour are among the most basic metaphorical representations for objects. This idea is in line with a number of findings in cognitive science, including the one that shape plays a fundamental role in kind classification [41] (infants will tend to classify things as being of the same kind if they share a similar shape). Moreover, our most primitive notion of object (in fact, our most primitive sortal type) is the notion of a maximally-topologically-self-connected object which moves in a spatial temporal trajectory together with all its parts [42]. This idea is directly represented by convex shapes with closed boundaries.

In the language used in Fig. 10, each distinct concrete (sub)kind of domain objects is associated with a shape in the aforementioned sense. Moreover, the chosen shapes are sufficiently dissimilar and are aligned with the taxonomic relations between domain types as presented in Fig. 10. For example, the "four-sized" figures used to represent Organizational Branches and Departments are similar, respecting the fact that they are both direct subtypes of organizational units. On the other hand, they are dissimilar from the blobs used to represent Commissions and the Icons used to represent People. These features allow for another important quality characteristic discussed by Moody [36], namely, *Perceptual Discriminability*. For this reason, by looking at a model in this language one can immediately tell which domain element is being represented by which graphical element.

A final aspect worth mentioning is the direct metaphorical resemblance between the graphical elements used and their referents. The most obvious case is the iconic representation for Man and Woman. However, the representation of Departments as "pieces of an Organizational Branch" is adherent to the idea of "Organizational Divisions" associated with Departments, Sectors, etc. In addition, while the straight lines used in the contour of Organizational Units gives the idea of more formal and rigid structure, the round boundaries of the blob representing Commissions is more naturally associated with a flexible informal one. The systematic use of these metaphorical resemblances brings to this notational system another important quality characteristic according to Moody, namely, *Perceptual Immediacy* [36]. In this example, by looking at the icons used to represent Man and Woman, one can directly infer the type of referent which is being represented.

Phases: As previously discussed, phases represent contingent specializations of kinds such that the specialization condition is related to the changes of value in the intrinsic properties of the instances of that kind. Accordingly, here once more we use an intrinsic property of visual percept used to represent the kind to represent different phases associated with that kind, i.e., the entity can be seen as changing phases but maintaining its identity due to the persistence of the visual percept. In the example of the language used in Fig. 10 the changes in color of the Blob used to represent Commissions represent different phases of a Commission. Moreover, we use a high-saturation color to represent the *Work-Overloaded Commission* exploring a metaphorical relation between "more quantity of color" (which is the definition of saturation) and "more quantity of work." Once more, this feature of the graphical grammar increases its *Perceptual Immediacy*. Moreover, the difference in brightness of the gray hue used to represent an overloaded commission, on the one side, and the white one used to represent a regular load commission on the other creates an efficient *Perceptual Pop-Out* [36], decreasing the cognitive cost of identifying former types of commissions in visual queries [40]. Finally, given that identifying overload commissions is an important task in this domain, the *perceptual pop-out* at hand is increased by the increased *perceptual discriminability* between these two phases. This is due to the use of a different in thickness of the blob boundaries. This is a case of what is termed *Dual Coding* in Moody's work [36] and it constitutes a small deviation of Semiotic Clarity for the specific purpose of increasing efficiency in particularly important visual queries.

Relations: In the ontology of Fig. 6, we have three types of relations. Firstly, we have the relations of *component-of* parthood between (1) Employee and Department, and (2) Department and Organizational Branch. As discussed in [33], *componentOf* is irreflexive and asymmetric. Moreover, transitivity holds across (1) and (2). By using the relation of spatial inclusion in the plane to represent these relations, we have a mapping to a visual relation that has exactly the same formal properties of the represented one, since spatial inclusion is also a partial order relation. This feature of the visual notation allows for a direct *inferential free ride* [43]: when identifying some as being *part of* the Marketing Department in Organizational Branch A, we immediately identify this person as being *part of* Organizational Branch A.

A second aspect that we would like to point out is that the different departments that comprise an Organizational Branch are represented by a tessellation of the spatial region used to represent that Branch. The lack of overlap between these regions allows for a *perceptually immediate* representation of the non-shareability meta-property of these *component-of* relations. In other words, since Departments are represented by tessellations (with non-overlapping regions), it becomes perceptually immediate to the user of the language that Employees are part of at most one Department and that Departments are parts of at most one Organizational Branch. This representation also contributes to *perceptual immediacy* due to yet another reason, namely, that if Departments are represented as *partitions of the region* representing its associated Branches, this also favors the interpretation of existential dependence (inseparability) from the part to the whole. To put it in a different way, it is much easier to visualize the icon for Man and Woman moving in and out of the Branch region than to visualize a piece of the region moving to another region.

Another type of parthood relation used in Fig. 6 is the one between Commission Member and Commission. Once more, this relation is also represented as a spatial containment relation between the icons representing People and the blob representing the Commission. However, as one can notice in Fig. 10, these blob forms can overlap with Branch and Department regions. This feature allows for the direct inferential free ride on the identification of which departments and branches commission members belong to. In addition, in line with the *shareability* meta-property of this relation in the ontology, one can easily imagine overlapping blobs allowing for a certain member to be simultaneously part of multiple commissions.

A third relation in this ontology is the *reports to* relation, defined between a (superior) employee and its subordinates. As previously discussed, this relation also defines a partial order relation between Employees. Here, we used a combination of visual relations to represent this domain association, namely, we combined the *above* relation in the plane (which is a total order relation) with the transitive closure of the *is-dashed-line-connected* relation. The combined relation is also a partial order relation. Additionally, the different texture of this line increases the *Perceptual Discriminability* when contrasting it to the solid lines used to demarcate department partitions. Finally, the spatial metaphor of using "higher in the plane" to represent "higher in the hierarchy" favors *Perceptual Immediacy* in this representation.

Roles and Relational Properties: roles represent contingent specializations of kinds which, in contrast with phases, have a relational specialization condition, i.e., a role R_1 is played by entities of type A when associated via a certain relation T to entities of type B, typically playing a role R_2 [17]. Accordingly, roles R_1 and R_2 should be represented by a visual relation between the visual representations of A and T, and B and T, respectively. This strategy has been employed consistently for the representation of all roles in this visual notation. For example, the role of Employee is represented by the *contained in region* relation between a People icon and the region representing the Branches. *Mutatis Mutandis*, the same can be said for the role of Commission Member. Finally, the complementary roles of *Superior* and *Subordinate* are accordingly represented by the adjacency relation between the People icons and the terminations of the dashed line representing the *reports-to* relation. The otherwise symmetric feature of this line (possibly suggesting a symmetric relation) is broken by the above relations between the icons in the plane. This asymmetry could be highlighted by the use of an asymmetric connector line. However, since asymmetry is already guaranteed by the combined relation, and our visual cognition is particularly efficient to spot vertical misalignment, we advise against such a design choice. Especially since this choice not only would hurt *Semiotic Clarity* but also what Moody terms *Graph Parsimony*. This is captured in the following quote from [36]: *"Empirical studies show that increasing graphic complexity significantly reduces understanding of software engineering diagrams by naïve users . . . It is also a major barrier to learning and use of a notation"*.

In Table 1 below, we present a correspondence between the concrete elements in the ontology of Fig. 6 and the system of graphical symbols comprising a visual modeling language used in Fig. 10.

6 Final Considerations

In this chapter, we elaborate on the relation between a modeling language and a set of real-world phenomena that this language is supposed to represent. We focus on two aspects of this relation, namely, the *domain appropriateness*, i.e., the suitability of a language to model phenomena in a given domain, and its *comprehensibility appropriateness*, i.e., how easy it is for a user of the language to recognize what that language's constructs mean in terms of domain concepts and how easy it is to understand, communicate, and reason with the specifications produced in that language. We defend that both of these properties can be systematically evaluated for a modeling language w.r.t. a given domain in reality by comparing a concrete representation of the worldview underlying this language (captured in a meta-model of the language), with an explicit and formal representation of a conceptualization of that domain, or a *reference ontology*.

We therefore present a framework for language evaluation and (re)design which aims, in a methodological way, to approximate or to increase the level of homomorphism between a meta-model of a language and a reference ontology.

Table 1 Visual concrete syntax for the organization structure ontology of Fig. 6

Domain Type	Ontological Category	Notational Element
Person	Kind	Abstract Class; No direct representation
Man	Subkind	
Woman	Subkind	
Organizational Unit	Kind	Abstract Class; No direct representation
Organizational Branch	Subkind	
Department and Department is component of Organizational Branch	subkind and part-whole relation	
Comission	Kind	Abstract Class; No direct representation
Normal Load Comission	Phase	
Overloaded Comission	Phase	
Employee and Employee is component of Department	role and part-whole relation	
Superior	Role	
Subordinate Employee	Role	
Commission Member and Commission Member is part of Comission	role and part-whole relation	
SubordinateEmployee reports to Superior	Domain association	combination of is-dashed-line-connected with the above relation in the plane

This framework comprises a number of properties (*lucidity*, *soundness*, *laconicity*, *completeness*) that must be reinforced for an isomorphism to take place between these two entities. The framework proposed combines two existing proposals in the literature: (i) the one presented in [3, 4], which focuses on the evaluation of individual representations and (ii) the one of [5, 6], which aims at the evaluation of representation systems. In addition, our framework extends these two proposals

in several ways and, compared to them, our framework possesses a number of advantages discussed in the sequel.

The approach of Gurr uses regular algebraic structures to model a domain conceptualization. We strongly defend the idea that the more we know about a domain the better we can evaluate and (re)design a language for domain and comprehensibility appropriateness. As we show in this chapter, there are important meta-properties of domain entities (e.g., rigidity, relational dependency) that are not captured by ontologically neutral mathematical languages (such as algebras or standard set-theories) and that the failure to consider these meta-properties hinders the possibility of accounting for important aspects in the design of efficient visual pragmatics for visual modeling languages. By demonstrating how these ontological meta-properties could be used for these purposes, this chapter also contributes towards the construction of a systematic connection with the framework for the evaluation and design of concrete visual syntaxes proposed by Moody in [36, 37].

The approach of Wand and Weber focuses solely on the design of general conceptual modeling languages. The framework and the principles proposed here instead can be applied to the design of conceptual modeling languages irrespective to which generalization level they belong, i.e., it can be applied both at the level of material domains and corresponding domain-specific modeling languages, and at the (meta) level of a domain-independent (meta) conceptualization that underpins a general conceptual (ontology) modeling language. For the case of domain-independent meta-conceptualizations, the framework discussed here has been applied to a number of prominent approaches (e.g., UML, Archimate, Tropos/i*, AORML, BPMN, ARIS, RM-ODP) as mentioned in Sect. 4 of this chapter. For the case of domain-specific conceptualizations, this ontology-based framework amounts to an important contribution to the area of domain-specific languages design methodologies (as acknowledged, for instance, in [44–46]).

As in the original proposal of Wand and Weber, the focus of our framework is on the level of systems of representations, i.e., on the evaluation of modeling languages, as opposed to a focus on individual diagrams produced using a language. Nevertheless, as it is demonstrated here, by considering desirable properties of the mapping of individual diagrams onto what they represent, we are able to account for desirable properties of the modeling languages used to produce these diagrams, extending in this aspect Wand and Weber's work.

Finally, both the approaches of Gurr and Wand and Weber address solely the relation between ontological categories and the modeling primitives of a language, paying no explicit attention to the possible constraints governing the relation between these categories. Moreover, they do not consider the necessary mapping from these constraints to equivalent ones, to be established between the language constructs representing these ontological categories. As demonstrated in Sect. 3 of this chapter, the proposal presented here, in contrast, explicitly considers the constraints governing the relations between the elements comprising a given domain conceptualization, and how these constraints are taken into account in a representation system. In particular, we demonstrate here how a strategy based on

visual simulation can be used to validate this aspect of reference ontologies and the language meta-models based on them.

In summary, as a contribution for Domain Engineering, we presented a framework that can be used for the evaluation and (re)design of reference models, in general, and *Domain Models*, in particular. This approach can be employed to formally and systematically improve the quality of domain models that, in turn, can be used to derive the meta-models of domain-specific conceptual modeling Languages. However, the use of an ontology-based method for producing high-quality domain models which encompass consistent, comprehensive, and truthful domain axiomatizations is expected to have a significant impact on other domain engineering enterprises such as the principled design of domain frameworks [47].

Acknowledgments The author is greateful to Renata S.S. Guizzardi, João Paulo Almeida, Tiago Prince, Alessander Benevides, Bernardo Braga, Luís Ferreira Pires, Marten van Sinderen, and Nicola Guarino for the different lines of collaboration which are reflected in the topics of this chapter. In particular, he would like to express his gratitude to Gerd Wagner for the great number of insightful inputs over 10 years of fruitful collaboration. This research is supported by FAPES (PRONEX #52272362/2010) and CNPq (productivity grant #311578/2011-0).

References

1. Guizzardi, G., Ferreira Pires, L., van Sinderen, M.: An Ontology-Based Approach for Evaluating the Domain Appropriateness and Comprehensibility Appropriateness of Modeling Languages, ACM/IEEE 8th International Conference on Model Driven Engineering, Languages and Systems. LNCS, vol. 3713. Springer (2005)
2. Guizzardi, G.: On ontology, ontologies, conceptualizations, modeling languages, and (meta)models. In: Vasilecas, O., Edler, J., Caplinskas, A. (eds.) Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IV. IOS, Amsterdam (2007). ISBN 978-1-58603-640-8
3. Gurr, C.A.: Effective diagrammatic communication: syntatic, semantic and pragmatic issues. J Vis Lang Comput **10**, 317–342 (1999)
4. Gurr, C.A.: On the isomorphism, or lack of it, of representations, Chapter 10. In: Marriot, K., Meyer, B. (eds.) Theory of Visual Languages. Springer, Berlin (1998)
5. Wand, Y., Weber, R.: An ontological evaluation of systems analysis and design methods. In: Falkenberg, E., Lingreen, P. (eds.) Information System Concepts: An In-Depth Analysis. Elsevier, North-Holland (1989)
6. Weber, R.: Ontological Foundations of Information Systems. Coopers & Lybrand, Melbourne (1997)
7. Grice, H.P.: Logic and conversation. In: Cole, P., Morgan J. (eds.) Syntax and Semantics. Speech Acts, vol. 3, pp. 43–58, Academic, New York (1975)
8. Guarino, N.: Formal Ontology in Information Systems. Proceedings (FOIS), Italy. IOS (1998)
9. Fitting, M., Mendelsohn, R.L.: First-Order Modal Logic. Kluwer, Norwell (1999)
10. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. Universal Press, The Netherlands (2005). ISBN 1381–3617
11. Ciocoiu, M., Nau, D.: Ontology-Based Semantics, 7th International Conference on Principles of Knowledge Representation and Reasoning (KR'2000), USA, 2000
12. Wand, Y., Weber, R.: Mario Bunge's ontology as a formal foundation for information systems concepts. In: Weingartner, P., Dorn, G.J.W. (eds.) Studies on Mario Bunge's Treatise. Rodopi, Atlanta (1990)

13. Guizzardi, G., et al.: Grounding software domain ontologies in the unified foundational ontology (UFO): the case of the ODE software process ontology. In: 11th Ibero-American Conference on Software Engineering (CIbSE), Recife, 2008
14. Green, P., Rosemann, M.: Integrated process modeling: an ontological evaluation. Inf Syst **25**(2), 73–87 (2000)
15. Bera, P., Wand, Y.: Analyzing OWL using a Philosophy-Based Ontology, 3rd International Conference on Formal Ontology in Information Systems (FOIS 2004), Torino, 2004
16. Guizzardi, G., Wagner, G.: What's in a Relationship: An Ontological Analysis, 27th International Conference on Conceptual Modeling (ER'2008), Barcelona, Spain. Lecture Notes in Computer Science, vol. 5231, pp. 83–97 (2008)
17. Guizzardi, G., Wagner, G., Guarino, N., van Sinderen, M.: An Ontologically Well-Founded Profile for UML Conceptual Models, 16th International Conference on Advances in Information Systems Engingeering (CAiSE), Latvia. LNCS 3084. Springer-Verlag (2004)
18. Guizzardi, G.: Modal Aspects of Object Types and Part-Whole Relations and the de re/de dicto distinction, 19th International Conference on Advanced Information Systems Engineering (CAISE'07), Trondheim. LNCS 4495. Springer-Verlag (2007)
19. Guizzardi, G., Masolo, C., Borgo, S.: In Defense of a Trope-Based Ontology for Conceptual Modeling: An Example with the Foundations of Attributes, Weak Entities and Datatypes, 25th International Conference on Conceptual Modeling (ER'2006), Arizona, 2006
20. Azevedo, C., Almeida, J.P.A., van Sinderen, M.J., Quartel, D., Guizzardi, G.: An Ontology-Based Semantics for the Motivation Extension to ArchiMate, 15th IEEE International Conference on Enterprise Computing (EDOC 2011), Helsinki, Finland
21. Almeida, J.P.A., Guizzardi, G.: An Ontological Analysis of the Notion of Community in the RM-ODP Enterprise Language Original Research Article, Computer Standards & Interfaces. Elsevier (2012). doi:10.1016/j.csi.2012.01.007
22. Guizzardi, R.S.S., Guizzardi, G.: Ontology-based transformation framework from Tropos to AORML. In: Giorgini, P., Maiden, N., Mylopoulos, J., Yu, E. (Org.) Social Modeling for Requirements Engineering, Cooperative Information Systems Series. MIT Press, Boston (2010)
23. Guizzardi, R.S.S., Franch, X., Guizzardi, G.: Applying a Foundational Ontology to Analyze the i* Framework, 6th International Conference on Research Challenges in Information Systems (RCIS 2012), Valencia
24. Santos Jr., P.S.S., Almeida, J.P.A., Guizzardi, G.: An Ontology-Based Semantic Foundation for ARIS EPCs. In: Proceedings of the 25th ACM Symposium on Applied Computing (Enterprise Engineering Track), 2011
25. Guizzardi, G., Wagner, G.: Can BPMN Be Used for Simulation Models? 7th International Workshop on Enterprise & Organizational Modeling and Simulation (EOMAS 2011), together with the 23rd International Conference on Advanced Information System Engineering (CAiSE'11), London, UK
26. Guizzardi, G., Herre, H., Wagner, G.: On the General Ontological Foundations of Conceptual Modeling, 21st International Conference on Conceptual Modeling (ER), Finland. LNCS 2503. Springer-Verlag (2002)
27. Barcelos, P.P.F., Guizzardi, G., Garcia, A.S., Monteiro, M.E.: Ontological Evaluation of the ITU-T Recommendation G.805, 18th International Conference on Telecommunications (ICT 2011). IEEE Press, Cyprus (2011)
28. Guizzardi, G., Baião, F., Lopes, M., Falbo, R.: The role of foundational ontologies for domain ontology engineering: an industrial case study in the domain of oil and gas exploration and production. Int. J. Inf. Syst. Model. Design **1**(2), 1–22 (2010)
29. Gonçalves, B., Guizzardi, G., Pereira Filho, J.G.: Using an ECG reference ontology for semantic interoperability of ECG data. J. Biomed. Inform. **44**, 126–136 (2011)
30. Costal, D., Gomez, C., Guizzardi, G.: Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML, 30th International Conference on Conceptual Modeling (ER 2011), Brussels, 2011

31. Benevides, A.B., Guizzardi, G.: A Model-based Tool for Conceptual Modeling and Domain Ontology Engineering in OntoUML, 11th International Conference on Enterprise Information Systems (ICEIS), Springer, Milan, 2009
32. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
33. Guizzardi, G.: The Problem of Transitivity of Part-Whole Relations in Conceptual Modeling Revisited, Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAISE'09), Amsterdam, 2009
34. Benevides, A.B., Guizzardi, G., Braga, B.F.B., Almeida, J.P.A.: Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures. J Univ Comp Sci **16**(20), 2904–2933 (2010)
35. Jackson, D.: Alloy: a lightweight object modelling notation. Trans Software Eng Methodol **11**(2), 256–290 (2002)
36. Moody, D.L., van Hillegersberg, J.: Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams, 1st International Conference on Software Language Engineering (SLE), 2008, pp. 16–34
37. Moody, D.L.: The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans Software Eng **35**(6), 756–779 (2009)
38. Goodman, N.: Languages of Art: An Approach to a Theory of Symbols. Bobbs-Merrill, Indianapolis (1968)
39. Nordbotten, J.C., Crosby, M.E.: The effect of graphic style on data model interpretation. Inf Syst J **9**(2), 139–156 (1999)
40. Ware, C.: Visual Thinking for Design. Morgan Kaufmann (2008)
41. Tversky, B., Hemenway, K.: Objects, parts, and categories. J Exp Psychol Gen **113**, 169–193 (1984)
42. Xu, F., Carey, S.: Infants' metaphysics: the case of numerical identity. Cog Psychol **30**, 111–153 (1996)
43. Shimojima, A.: Operational constraints in diagrammatic reasoning. In: Logical Reasoning with Diagrams, pp. 27–48. Oxford University Press, New York (1996)
44. Becker, J., Pfeiffer, D., Räckers, M.: PICTURE – A NEW APPROACH for Domain-Specific Process Modelling, Proceedings of the CAiSE'07 Forum at the 19th International Conference on Advanced Information Systems Engineering, Trondheim, 2007
45. McDermott, J., Allwein, G.: A formalism for visual security protocol modeling. J Vis Lang Comput **19**(2), 153–181 (2008)
46. McDermott, J.: Visual Security Protocol Modeling, Proceedings of the 2005 Workshop on New security paradigms, New York, 2005
47. Falbo, F., Guizzardi G., Duarte, K.: An ontological approach to domain engineering. In: Proceedings of 14th International Conference on Software and Knowledge Engineering (SEKE). ACM, New York (2002)

Automating the Interoperability of Conceptual Models in Specific Development Domains

Oscar Pastor, Giovanni Giachetti, Beatriz Marín, and Francisco Valverde

Abstract An increasing number of modeling approaches for representing concepts related to a wide variety of domains can be clearly observed in software engineering. In this context, the definition of sound interoperability mechanisms to reuse knowledge and share ideas among existing conceptual models, and also apply them into concrete development processes, is an important challenge to be faced. Thus, different modeling approaches, tools, and standards can be integrated and coordinated to reduce the implementation and learning time of development processes as well as to improve the quality of the final software products. However, there is a lack of approaches to support automatic interoperability among modeling approaches. For tackling this situation, this chapter presents an interoperability approach focused on the characterization of different modeling approaches in a common software development domain. For putting in practice and automate the interoperability the approach proposed, existing modeling technologies and standards are considered. All these elements comprise a reference interoperability model, which can be used to implement specific interoperability solutions.

Keywords Conceptual modeling • Interoperability model • Interoperability process • Literature review • Model-driven interoperability

---

O. Pastor (✉) • F. Valverde
Centro de Investigación en Métodos de Producción de Software (PROS), Universitat Politècnica de València, Valencia, Spain
e-mail: opastor@dsic.upv.es; fvalverde@pros.upv.es

G. Giachetti
Facultad de Ingeniería, Escuela de Informática, Universidad Andres Bello, Sazié 2325, Santiago, Chile
e-mail: ggiachetti@unab.cl

B. Marín
Escuela de Ingeniería Informática, Facultad de Ingeniería, Universidad Diego Portales, Santiago, Chile
e-mail: beatriz.marin@mail.udp.cl

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_14, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

According to the IEEE Standard Computer Dictionary [44], interoperability is defined as *the ability of two or more systems or components to exchange information and to use the information that has been exchanged*.

Interoperability is a key aspect to be faced in many different areas where interchange of information and knowledge is necessary to achieve the intended objectives. This is a well-known fact in the electronic device world, where interoperability is critical for positioning and commercializing new technologies. Nobody thinks of changing the DVD player depending on the DVD brand to play.

Other example can be found in the architecture field, where interoperability is essential for communicating building information among the different actors involved, i.e., among architects, engineers, and finally to the builders, which become the charts specified into concrete and physical structures. This last situation is very close to the software domain reality, where the correct representation of concepts that must be implemented is essential to obtain (build) software products that fit with customer's needs. In the domain engineering world, the definition of suitable conceptual elements related to a specific development domain is the Rosetta stone for depicting and obtaining appropriate software models (software charts).

The necessity of counting with interoperability mechanisms in the current software development context is a growing trend. For instance, we can observe interoperability in web applications where different web services must be coordinated to perform a specific operation. Also, interoperability becomes necessary for geographically distributed software factories that are developing different components of a same software product [16].

However, the interoperability of conceptual models presents different unsolved challenges, such as the definition of concrete mechanisms for automation and verification of interoperability operations. Moreover, models defined, even with a same modeling language, are not compatible among different modeling tools. UML is a clear example of this situation. Probably, the large variety of modeling approaches and related technologies is mainly responsible for the lack of interoperability among conceptual models. Therefore, we need to deal with this model heterogeneity since the modeling representations that can be defined in the conceptual modeling domain are practically infinite.

Thus, in this chapter, we propose a conceptual model for the definition of concrete interoperability solutions, which deals with model heterogeneity aspects and faces automation issues. This interoperability model has been instantiated with current standard and model-driven technologies to obtain a suitable solution for concrete model-based development processes.

The rest of this chapter is organized as follows: Sect. 2 presents the related work in the context of model-driven interoperability. Section 3 details the interoperability model proposed. Section 4 presents the challenges that we have faced in the definition of this interoperability model. Section 5 introduces the process to put into practice the interoperability model. Finally, Sect. 6 presents a general analysis of the interoperability proposal presented and main conclusions.

2 Review of Model-Driven Interoperability

One of the main concerns of the domain engineering field is the definition of suitable constructs and conceptual models for performing the analysis of specific application domains [26]. In this context, the reuse of domain knowledge and its application into concrete software production process is a relevant issue to be considered. The use of conceptual models, domain-specific representations, and model reuse strategies that are part of the domain engineering principles are also subject of interest in the model-based software production context. For instance, the application of domain analysis approaches, which come from the requirements engineering world, into MDD processes [5, 53, 68].

In this chapter, we face the interoperability problem from the model-driven perspective, thus proposing a *Model-Driven Interoperability* approach; i.e., the information exchange involved in interoperability tasks is performed by means of models and model management operations (such as model transformations).

In order to explore the existing evidence of model-driven interoperability, we have reviewed relevant approaches related to different domains, such as conceptual (including domain-specific) modeling, requirement engineering, and business process modeling. [35].

In the revision, the following characteristics are considered:

- The proposal provides mechanisms for *managing the heterogeneity* that may exist among the involved modeling approaches.
- The proposal considers the use of existing *standards* for the definition of models, metamodels, or model transformations.
- The proposal has some kind of *supporting technology* that automates the interoperability operations.
- The proposal provides a well-defined *application process*.
- The proposal defines *mechanisms to verify the interoperability* among the modeling approaches involved.

The studies analyzed are briefly presented as follows. We have grouped the studies according to their main features (some approaches provide more than one relevant contribution), which correspond to the following:

- *Model Weaving* (*MW*): It considers the definition of mappings among the metamodels of involved modeling approaches. These mapping are usually defined by means of a weaving model (instance of a weaving metamodel) [25].
- *Meta-Extensions* (*ME*): This corresponds to the definition of new information (extensions) in the modeling approaches involved to provide additional modeling features that are necessary to perform interoperability operations [77].
- *Interoperability Verification* (*IV*): Models and interoperability operations may have issues that produce an incomplete interchange of information. Thus, it is important to analyze verification mechanisms for assuring the correct specification of the artifacts that are involved in interoperability scenarios [34].

- *Pivot Artifact* (*PA*): This feature is related to the use of an intermediate artifact (metamodel or ontology) to manage structural differences and to identify conceptual equivalences [42].
- *Application Domain* (*AD*): Indicates the domain that is related to the approach analyzed. Despite this, the contributions of the proposal analyzed can be generalized to different domains.

2.1 Model Weaving

The model weaving approach is very popular in model transformation and model interchange contexts. Weaving models indicate semantic equivalences through the definition of links among the metamodels' constructs. These links are considered as semantic connection points because they indicate those constructs (from the involved metamodels) that have an equivalent meaning (semantics) in the application domain. The definition of these links can be extended with additional information defined to manage structural differences among the linked constructs.

Fabro and Valduriez in [25] propose the use of weaving models between two metamodels to automatically infer model-to-model (M2M) transformations based on the ATL tool [46]. These transformations automate the translation between models defined with the involved metamodels.

The proposal presented in [49] by Kappel et al. is defined to support the interoperability among modeling tools. This proposal is based on a bridge metamodel (weaving metamodel) for the definition of semantic metamodel links (bridges). The defined bridges are used to transform the involved metamodels into equivalent petri-net representations. The petri-net representation is used to operationalize M2M transformations and to perform a formal verification of the structural differences (heterogeneities) among metamodels, which may produce interoperability conflicts.

The proposal presented by Klar et al. in [50] shows how the MDD interoperability can be used to support a complete development process. In particular, this proposal is centered on the integration of requirements modeling into MDD processes. However, it does not consider how to integrate specific aspects related to a particular MDD process into the requirements approach. These MDD aspects are necessary to obtain an appropriate requirement specification in the domain of the reference MDD approach.

The proposal presented in [36] by Guerra et al. consists of a pattern-based approach for defining bidirectional relations (a weaving model) among modeling approaches. The main contributions of this proposal are the definition of specific inter-modeling patterns, which allow interoperability conflicts to be automatically identified. These patterns also facilitate the generation of model-to-model transformations, model matching, and traceability information.

2.2 Meta-Extensions

Seifert et al. in [75] indicate the advantages of using metamodels and model-to-model transformations to prevent the coupling among tools that must interoperate. This approach analyzes the pros and cons of proactive and retroactive tool integration alternatives. From this analysis, it suggests the use of a role-based metamodeling approach, which involves the extension of the metamodels of tools with specific role information (defined in a role metamodel) to improve the tool interoperability.

The *BIZYCLE* framework applied by Milanovic et al. in [58] is defined to achieve applications and data integration by means of semantic annotations. These annotations are used to identify both semantic and structural conflicts that can be solved in a semi-automatic way.

The work by Tran et al [79] suggests the extension of the modeling approaches that must interoperate to specify the information related to an MDD process.

The proposal presented by Agostinho et al. in [4] introduces an interoperability framework for business networks, which is based on UML for the definition of the involved metamodels. An interesting feature of this approach is the use of UML profiles to manage model heterogeneities and to obtain an appropriate model mapping. This work refers to those transformations that imply a structural change of the involved constructs as *model altering morphisms*.

2.3 Interoperability Verification

Radjenovic and Paige in [70] present an interoperability approach that is based on an initial identification of the issues that may prevent an appropriate model integration. This work considers both structural and behavioral interoperability conflicts. The detection of interoperability issues is performed by means of the transformation of the involved metamodels into a proprietary graph representation, which is called SMILE-X.

The proposal presented by Polgár et al. [69] indicates the need for interoperability in a common development process. In this approach, a reference ontology is used to verify whether or not the involved modeling approaches are in conformance with the target development process.

2.4 Pivot Metamodel

The differences between a pivot metamodel and a weaving metamodel are related to their definition and use. A weaving metamodel is instantiated to represent links among constructs of the involved metamodels. From these weaving models, specific M2M transformation rules can be inferred according to a specific transformation

approach, such as ATL [46], QVT [63], or ETL [51]. However, weaving models do not participate in the execution of the transformation rules. By contrast, a pivot metamodel can be a predefined representation of concepts (or constructs) for the interoperability domain or can be generated from the metamodels of the modeling approaches that must interoperate. Also, a pivot metamodel can be instantiated during the transformation process generating an intermediate pivot model.

The proposal presented in [16] by Bruneliere et al. defines a pivot metamodel to solve conflicts related to the heterogeneity among metamodels of modeling tools. This proposal is also based on current metamodeling standards and modeling tools.

The DUALLY approach presented by Crnkovic et al. [19] shows that the use of a pivot metamodel reduces the complexity of the necessary transformation rules. These rules can be automatically inferred from the pivot metamodel definition.

The proposals presented by Ziemann et al. [86] and Jankovic et al. [45] are related to enterprise modeling. They use the POP* metamodel [60] as pivot metamodel. According to these proposals, the involved modeling approaches must be mapped to the POP* metamodel to determine common interrelation points. Similar approaches are presented by Baumgart [9] in the domain of embedded systems and by Mahé [55] for visualization tools.

Berger in [10] considers the definition of a pivot metamodel that comprises all the conceptual constructs related to the modeling approaches that must interoperate. This pivot metamodel (defined as *generic metamodel* in the paper) is used as an interface among the metamodels of the involved modeling languages, which isolate the mappings from the metamodel heterogeneities. Later, by means of a set of predefined patterns, a model weaving among the involved metamodels is automatically generated to perform model-to-model transformations.

Vallecillo in [80] proposes the generation of a global model for the combination of different modeling approaches. The generation of this global metamodel is based on a viewpoint unification, which intends to comprise the benefits related to three metamodel integration techniques: metamodel extension, metamodel merge, and language embedding. The use of a common model obtained by the integration of the involved modeling approaches is also presented in Coutinho et al. [18]. This proposal is related to organizational modeling.

In [59], Moreno and Vallecillo propose a web development interoperability framework, which is centered on a generic metamodel for web development methods. Thus, by means of QVT transformations, the modeling approaches related to the different development method must be mapped to the reference metamodel. Evidently, due to the use of QVT, the proposal requires that the involved modeling approaches be defined in Meta-object-Facility (MOF)-compliant metamodels.

Diskin et al. [20] propose the generation of a pivot metamodel, which only indicates overlaps among different modeling approaches. This *overlap metamodel* reduces the complexity related to the definition of a big metamodel that covers all the modeling constructs of the involved modeling approaches. However, this proposal is still theoretical and is not supported by tools or standards.

In the work presented by Biehl et al. [12], the relevance of defining a bridge between technical spaces is clearly stated. This technical bridge is oriented to

translating the metamodels of the involved modeling tools into equivalent representations that are defined using a common metamodeling language, i.e., this solves technical interoperability conflicts. Later, structural bridges are defined in the common interoperability space to perform M2M transformations among the metamodels generated by the technical bridges. A similar approach is defined by Jouault and Guéguen in [47]. In this approach, concrete modeling tools are translated into equivalent metamodeling representations, which are defined with a common metamodeling language. The resultant metamodel is called *virtual tool*. A similar view is presented by Brambilla et al. [14], whose work proposes the translation of domain-specific languages (DSL) to equivalent MOF representations.

In addition, in a previous work presented by Lukácsy et al. in [54], the outputs generated by different information sources are transformed into equivalent models (called *interface models*) to perform interoperability among web services. These interface models are expressed in a UML-like language. An improvement to this kind of service-oriented works is presented by Tran et al. in [79]. In this paper, the authors propose a reverse-engineering mechanism to automatically infer model representation from services' views. The inference models are integrated in a view-based modeling framework to perform integration of services. These models are also used to generate code in other implementation platforms by following an MDD process.

The *ModelBus* approach presented by Hein et al. in [40] is oriented to tool interoperability among nodes that participate in a common development scenario. The interoperability is performed by means of a repository of models and modeling services (such as model transformation and model verification services). This approach is based on the original idea of model bus presented in [13], which indicates two important aspects that must be considered to achieve the MDD interoperability: the *functional connectivity* (related to metamodel heterogeneity) and the *protocol connectivity* (related to technical heterogeneity).

2.5 Pivot Ontology

The main difference between pivot ontologies and pivot metamodels is related to their application context and implementation aspects. While a pivot metamodel directly links syntactic elements (abstract syntax), a pivot ontology is based on semantic principles and requires a reference ontological standard, such as OWL and RDFS [49]. These standards only have a partial correspondence with metamodeling facilities (like MOF). Thus, a *lifting* process [48] is required to solve the gap between semantic (ontologies) and syntactic (metamodel) specifications.

Höfferer in [42] presents an interesting analysis related to the use of ontologies and metamodels to achieve the model-driven interoperability. Even though this work is framed in the context of business-process modeling, the analysis and conclusions obtained can be generalized to any model-interoperability approach.

The Sunindyo et al.'s proposal [78] uses a common bus to perform the model-driven interoperability (such as in [40]). This proposal uses ontology mappings to identify common semantic links among different modeling approaches; the definition of these links is guided by a process for automatic discovery of the involved process models.

Roser and Bauer present in [71] an approach that uses an ontology specification (based on OntoMT) as an intermediate model for managing the heterogeneities and similarities among the metamodels of the involved modeling languages. It is also used to reuse the information of already defined M2M transformations and to reduce the complexity related to changes in the versions of the involved metamodels. This approach distinguishes two kinds of model transformations: (1) mappings, which are horizontal model transformations defined at the same abstraction level and (2) refinement transformations, which imply a change from a higher (less detailed) abstraction level to a lower (more detailed) abstraction level.

Berre et al. presents an ontology-based approach related to service interoperability in [11]. Barnickel and Fluegge proposes the idea of sematic mediation at the domain level to improve the efficiency and the effectiveness of the ontology-based interoperability in [8]. The semantic mediation defines a pivot ontology for each involved domain, which groups a set of conceptual schemas. According to these authors, this approach provides a balanced interoperability solution, which is at a middle point between defining ontologies and mappings for each conceptual schema and the definition of a common pivot ontology.

Opdahl in [66] presents a modeling approach that is framed in the context of business processes. It facilitates language interoperability by applying the unified enterprise modeling language (UEML) [43]. This approach requires the translation of the involved DSMLs into the equivalent UEML representation.

Also in the context of business processes, the proposal presented by Costa et al. [17] provides a model-based platform for the enterprise interoperability. This proposal uses a reference ontology to identify semantic equivalences among the information (messages) that must interoperate. This information (defined in an XML format) is extended with annotations to manage heterogeneities in relation to the reference ontology, which are used to perform appropriate model-to-model transformations.

Table 1 summarizes the works analyzed and the main characteristics that each work presents, which correspond to the following: Management of heterogeneity (MH), Use of Standards (US), Tool Support (TS), Application Process (AP), Interoperability Verification (IV), Meta-Extensions (ME), Pivot Artifact (PA), and Application Domain (AD). In the table, letters *Y* and *N* mean *Yes* and *No*, respectively. In the *PA* column, the letter *O* means *Ontology* and the letter *M* means *Metamodel*. According to the review results, the use of a common interoperability space with a unique *Metamodeling Specification* (*MS*) is an aspect that is considered by all the approaches analyzed. Furthermore 24 approaches, which correspond to 72.7 % of the total approaches analyzed, use *Pivot Artifact* (*PA*) for managing structural heterogeneities. This pivot artifact can be defined through a metamodel or an ontology. We recommend the use of pivot metamodels since the definition

Table 1 Summary of reference model-driven interoperability works

Author	Year	MW	MH	US	TS	AP	IV	ME	PA	AD
Agostinho [4]	2011	Y	Y	Y	N	N	N	Y	N	Bussiness networks
Barnickel [8]	2010	Y	Y	N	N	N	N	N	O	Services
Baumgart [9]	2010	N	Y	N	N	N	N	N	M	Embedded systems
Berger [10]	2010	Y	Y	N	Y	N	N	N	M	Conceptual modeling
Berre [11]	2009	Y	Y	Y	N	N	N	N	N	Services
Biehl [37]	2010	Y	Y	Y	Y	N	N	N	M	Tool interoperability
Brambilla [14]	2008	Y	N	Y	Y	Y	N	N	N	Migration DSL to MOF
Brunelière [16]	2010	Y	Y	Y	Y	Y	N	N	M	Modeling tools
Costa [17]	2007	N	Y	N	Y	N	N	Y	O	Business processes
Coutinho [18]	2009	Y	Y	Y	Y	Y	N	N	M	Org. modeling
Crnkovic [19]	2009	Y	N	Y	Y	Y	N	N	M	Component models
Diskin [34]	2010	Y	Y	N	N	Y	Y	N	M	Conceptual modeling
Fabro [25]	2009	Y	N	N	Y	N	N	N	N	Conceptual modeling
Guerra [36]	2011	Y	Y	N	Y	Y	Y	N	N	Conceptual modeling
Hein [40]	2009	N	Y	Y	Y	Y	N	N	M	Tool interoperability
Höfferer [42]	2007	Y	Y	N	N	N	N	N	O	Business processes
Jankovic [45]	2007	N	N	N	N	Y	N	N	M	Enterprise modeling
Jouault [47]	2009	Y	Y	N	N	Y	N	N	M	Tool interoperability
Kappel [49]	2011	Y	Y	Y	Y	Y	Y	N	N	Conceptual modeling
Klar [50]	2008	Y	N	Y	Y	Y	N	N	N	Requirement engineering
Lukácsy [54]	2007	N	Y	N	Y	N	N	N	M	Services
Mahé [55]	2010	Y	Y	N	Y	N	N	N	M	Visualization tools
Milanovic [58]	2009	Y	Y	Y	Y	Y	Y	N	O	Conceptual modeling
Moreno [59]	2008	N	Y	Y	Y	N	N	N	M	Web development tools
Opdahl [66]	2010	Y	Y	N	Y	N	N	N	O	WebML and UML
Polgar [69]	2009	N	Y	Y	Y	Y	Y	N	O	Model-driven development
Radjenovic [70]	2010	N	Y	N	Y	Y	Y	N	N	Conceptual modeling
Roser [71]	2007	Y	Y	N	Y	N	N	N	O	Conceptual modeling
Seifert [75]	2010	Y	N	N	N	Y	N	Y	N	Tool interoperability
Sunindyo [78]	2010	N	Y	N	Y	Y	N	N	O	Signal engineering
Tran [79]	2008	N	Y	Y	Y	Y	N	Y	N	Services
Vallecillo [80]	2010	Y	Y	N	N	N	N	Y	M	Conceptual modeling
Ziemann [86]	2007	N	Y	N	Y	Y	N	N	M	Enterprise modeling

of weavings among metamodels and ontologies (also called lifting [51]) implies additional complexity.

In relation to *Interoperability Verification* (*IV*) mechanisms, only six (6) approaches (18.2 % of the approaches analyzed) consider some kind of verification mechanisms. All these works are focused on verifying interoperability at *specification level*, which corresponds to appropriate specification of interoperability artifacts (weavings, pivots, and model transformations). However, none of the analyzed approaches consider the verification of the interoperability execution,

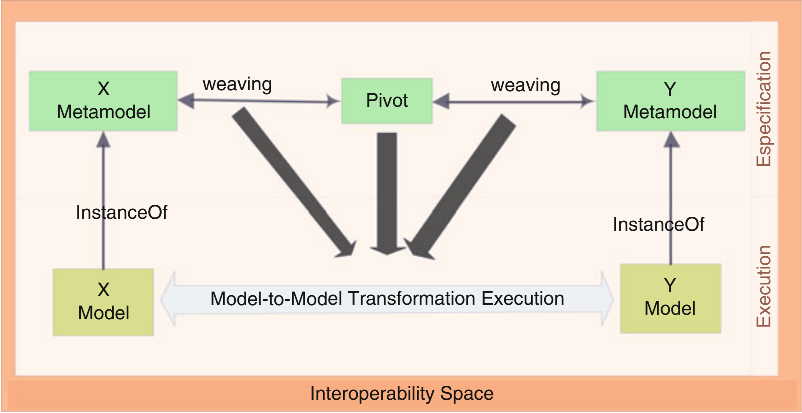


Fig. 1 General model-driven interoperability schema

i.e., the proper instantiation of the involved metamodels, and execution of model transformations.

Figure 1 provides a general model-driven interoperability schema obtained as results of this analysis.

Next section shows an interoperability model and the challenges that we have faced to achieve the automatic model-driven interoperability. This interoperability model is based on the model-driven schema obtained from the literature review results.

3 An MDD Interoperability Model

Model-based proposals related to the context of information systems and tool interoperability state different levels [39] to achieve an appropriate interoperability framework, such as [1, 67, 72, 83]. These proposals have several common aspects and present similar interoperability levels. In particular, we have centered our attention on the LISI (Levels of Information Systems Interoperability) [1] and LCIM (Levels of Conceptual Interoperability Models) [83] since the conceptual basis behind these two approaches covers the different features provided by previously analyzed studies. Moreover, these approaches can be easily generalized to different domains, have achieved a suitable maturity level, and their applicability has been empirically demonstrated.

The LISI approach was created by the U.S.A. defense department as a solution for defining, evaluating, measuring, and assessing information systems interoperability [1]. In this context, the correct and secure flow of information among the different systems is a critical concern. The LISI approach proposes an interoperability model comprised of five levels (from 0 to 4). Level 0 (isolated interoperability) corresponds to a manual interoperability, where the interoperation tasks must be

Table 2 The LISI reference model

Interoperability	Computing environment	Level	P	A	I	D
Enterprise	Universal	4	Enterprise level	Interactive	Multi-dimensional topologies	Enterprise model
Domain	Integrated	3	Domain level	Groupware	World-wide network	Domain model
Functional	Distributed	2	Program level	Desktop automation	Local networks	Program model
Connected	Peer-to-peer	1	Local/site level	Standard system drivers	Simple connection	Local
Isolated	Manual	0	Access control	N/A	Independent	Private

performed manually by the system users. Level 4 (enterprise interoperability) indicates that data and services are automatically interchanged by different applications in a transparent way for the system users.

The levels defined in the LISI reference model are transversally divided into four interoperability attributes called PAID, which correspond to Procedures, Applications, Infrastructure, and Data. Table 2 summarizes the LISI reference model.

The LCIM approach is related to modeling and simulation, and, hence, it is closer to the model-driven development domain. Modeling aspects related to LCIM have a direct correspondence to the modeling tasks involved in MDD processes. Simulation corresponds to the execution of the modeled systems; therefore, it can be considered equivalent to the model compilation tasks that are involved in MDD processes.

The LCIM proposal states seven interoperability levels (from 0 to 6). Level 0 corresponds to the non-interoperability level (the same as the LISI proposal). Level 6 corresponds to the conceptual interoperability. In this level, interoperability is achieved by means of the definition of mappings among the conceptual models that describe the involved systems. In other words, conceptual interoperability is achieved through the meta-specification of the software systems. Figure 2 shows the levels defined for the LCIM model.

If we project the LCIM ideas to the model-driven context, we can state that to achieve conceptual model-driven interoperability, it is necessary to define mappings among the involved modeling languages. To perform these mappings it is necessary to consider syntax, semantics, and technical aspects that are related to modeling languages definition, which corresponds to the levels 1, 2, and 3 of the LCIM approach.

The levels 4 and 5 (Pragmatic and Dynamic interoperability) of the LCIM approach are related to operation of information systems and management of systems' data in time. In a specific model-driven context, such as model-driven

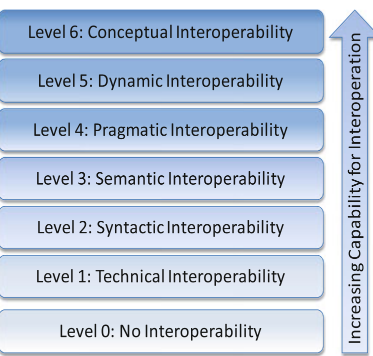


Fig. 2 LCIM model

development, these levels would be related to the evolution of the MDD models defined and their changes according to new system requirements. In the interoperability model that we propose, these interoperability levels are not considered since they are related to model synchronization and model evolution, which are topics that are not part of the model-driven interoperability vision. However, these are two interesting aspects that can be considered for future research in order to provide supporting facilities to model-driven interoperability.

In order to automate the model-driven interoperability, we have adopted the properties proposed by the LISI approach, which are the following:

- An appropriate interoperability *Procedure*, which indicates the elements that must be defined, and the steps that must be performed to interchange the modeling information.
- The *Applications* that manage the modeling information, which provide the features to automate interchange of models.
- The interoperability *Infrastructure*, which is related to the communication mechanisms among applications to assure the correct interchange of information and to prevent the loss of modeling information when the interchange process is performed.
- The *Data* (modeling information) must be specified in a standard format, which can be interpreted by different modeling tools with independency of implementation platforms and development contexts.

In summary, the interoperability model defined states model-driven interoperability in terms of technical, semantic, and syntactic interoperability. Also, model-driven interoperability can be automated by providing a concrete solution for procedure, application, infrastructure, and sata properties.

In relation to syntactic interoperability, different approaches have defined a particular syntax (abstract and concrete) to represent their modeling elements (conceptual constructs). The best example of this is UML [64]. This syntax is focused on supporting the semantics [38] of the modeling languages involved.

For the specification of the abstract syntax, it is possible to find standardized approaches, such as the MOF [62]. The MOF approach provides suitable support

for the generation of model-oriented technologies, such as model editors, and model transformation tools. This abstract syntax specification is performed by means of a metamodel definition, which represents the conceptual constructs (with their properties), the relationships that exist among the constructs, and a set of rules to manage the constructs' interaction.

From the metamodels that formalize the abstract syntax of modeling languages, the concrete syntax can be specified by using tools such as the eclipse graphical modeling framework (GMF) [23]. However, a standard for defining the concrete syntax related to a modeling language has not yet been defined.

The semantics related to modeling approaches is usually specified by means of textual representations, for instance, the UML specification [64] and the *i** framework [2]. In an MDD approach, we consider that the semantics is implicit in the mappings defined between the conceptual constructs and the corresponding software representations, which are used to perform the model-compilation process. However, there is a lack of an appropriate standard for the definition of the semantics related to modeling languages.

Thus, since only the abstract syntax of modeling languages is supported by standards that can be computationally interpreted, we propose the metamodels that formalize this abstract syntax as the starting point to support model-driven interoperability processes. From these metamodels, specific mappings can be defined (among the conceptual constructs of the involved modeling languages) to obtain semantic interoperability.

Technical interoperability can be achieved by using the interchange format defined for the open-source implementations of the metamodeling tools. For instance, the interchange mechanism implemented for the Eclipse UML2 tools is based on the XML specification [82]. This interchange mechanism is the XML Metadata Interchange (XMI) [61], which has been defined for the UML specification.

Thus, for automating model-driven interoperability it is necessary: (1) to establish an appropriate *Procedure* to generate the interoperability artifacts; (2) to indicate or implement the *Applications* that are necessary to manipulate the models and perform the model interchange; (3) to state the *Infrastructure* that will be used to communicate the *Applications*; and (4) to define the format used for the modeling *Data* representation.

We have chosen open-source applications to support automatic interoperability. In particular, we have considered the modeling tools developed in the context of the eclipse modeling project [22], such as the eclipse modeling framework (EMF) [21] and the Eclipse UML2 Project [24]. For the infrastructure, we have considered the XML implementation for the EMF tools, and the XMI specification related to the Eclipse UML2 models. The EMOF (EMF) and UML (Eclipse UML2) specifications provide the formats that are used to represent the modeling data. Thus, the applications, infrastructure, and data format are supported by current technologies, tools, and standards.

However, there is no standard procedure that can be used to perform automatic model-driven interoperability. Therefore we have defined a particular process that

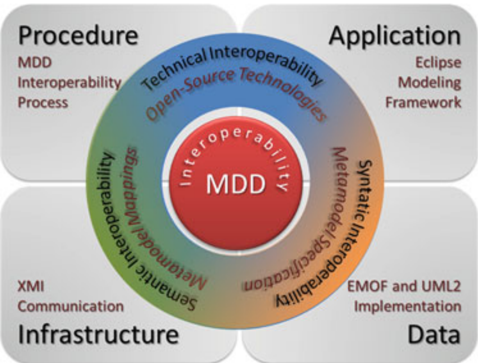


Fig. 3 MDD interoperability model instantiated

instantiates the proposed interoperability model to obtain an automatic model-driven interoperability solution. Figure 3 shows the interoperability model instantiated according to the analyzed standards, tools, and related works.

The process proposed for instantiating the interoperability model is mainly focused on the automatic interchange of modeling information in the context of an MDD process. In particular, it considers the integration of the modeling needs related to a specific MDD approach into already existing modeling approaches. For an appropriate representation of the specific MDD constructs, the constructs of the target modeling languages are customized to fix differences or to adding new properties in the context of the MDD approach. This is to define modeling language extensions. For the implementation of the required modeling language extensions, we use the UML profile extension mechanism since it is a standardized extension mechanism that has been improved according to the UML experience, and it is based on the MOF metamodeling standard. Therefore, the fundamentals related to UML profile extensions can be generalized to any modeling language that uses an MOF metamodel to formalize its abstract syntax. In addition, UML profile is a lightweight extension mechanism that does not alter the target metamodel, and, hence, the defined extensions do not affect the compatibility with the technologies that are based on the original specification of the modeling language customized.

By analyzing the previous background and related work, three main challenges must be faced to properly support model-driven interoperability. These challenges and the solutions proposed to solve them are detailed in the next section.

4 Challenges for Achieving the Model-Driven Interoperability

The first of the three challenges that must be solved for modeling language integration is to indicate the modeling artifact that will be used as starting point for this integration. The second challenge is related to define an appropriate mechanism to indicate the semantic equivalences between the involved modeling languages,

and the resolution of those interoperability issues that may prevent the correct interchange of modeling information. Finally, the third challenge is to automate the generation of the required metamodel extensions in order to reduce the potential errors and complexity that a manual modeling language customization involves.

4.1 First Challenge: Establish the Starting Point

For solving this first challenge, we have considered the definition (or selection) of the metamodels that are related to the involved modeling languages as starting point. These metamodels are the artifacts where equivalences among the modeling languages can be identified and the required extensions can be defined.

Metamodels provide good support to formalize the abstract syntax of modeling languages, which is essential to perform an appropriate integration of modeling languages. Also, in the current MDD context, metamodels are widely used for development of technologies and modeling languages. Thus, it has sense to consider an element that is commonly used by MDD-oriented approaches as starting point of an MDD interoperability process.

The paper presented by Selic in [76] indicates a set of elements that must be considered for an appropriate metamodel specification. These elements are the following:

- The set of conceptual constructs related to the modeling language, which are defined as classes (metaclasses) of the metamodel.
- The set of relationships that exist among the different conceptual constructs.
- The set constraints that manage the interaction among the different conceptual constructs, which are necessary to define valid models (instances of the metamodel).
- The notation related to each conceptual construct when corresponds.
- The meaning of the conceptual constructs defined.

For the specification of the involved metamodels we propose the use of the MOF metamodeling standard [62]. The use of MOF facilitates the definition of UML profiles for the implementation of modeling language extensions. Also, MOF is a suitable alternative for the specification of the required metamodels due to the following reasons:

- MOF is supported by a standardized interchange format (XMI [61]) .
- There exist different open-source metamodeling tools based on the MOF specification such as the Eclipse projects EMF [21] and UML2 [24].
- MOF is used by current model-to-model transformation technologies such as ATL [46] or QVT [63]
- There are many metamodel specifications based on MOF that can be used as reference modeling approaches.

- The use of MOF as common metamodeling language prevents the notation inconsistencies (at metamodel level) and facilitates the identification of equivalences between the different constructs.

However, there is an important lack in the MOF specification, which is the impossibility of indicating the notation (concrete syntax) and meaning (semantics) of the defined constructs [38]. The MOF metamodels only specify the abstract syntax of the corresponding modeling languages. Therefore, the notation and meaning of the constructs must be documented in a separated way. This information is relevant for the correct metamodel specification and it is helpful to understand the defined metamodels. Also, the notation and semantics are relevant for the appropriate implementation of MDD tools, such as modeling tools and model compilers.

The MOF specification provides two alternatives for metamodel definition, i.e., two metamodeling languages. The first of these languages is the complete set of constructs of the MOF specification, which is called CMOF (Complete-MOF). The second alternative corresponds to a subset of the MOF constructs, which provide essential metamodeling facilities. This second metamodeling language is called EMOF (Essential-MOF).

The metamodeling capabilities that are provided by EMOF are closer to the extension capabilities provided by UML profiles. By contrast, CMOF provides a set of metamodeling facilities that cannot be represented by means of UML profile extensions, for instance, n-ary associations, or property redefinition. Therefore, we consider the use of EMOF to specify the metamodels of the involved modeling languages.

Once the corresponding EMOF metamodels are specified, or selected in the case of already existing EMOF metamodels, the equivalences between metamodels must be indicated. These equivalences are used to identify the necessary metamodel extensions. At this point, the second challenge that must be faced arises.

4.2 Second Challenge: Identify Semantic Equivalences and Solve Interoperability Issues

This challenge involves the appropriate identification of semantic equivalences between the constructs related to a source and a target modeling language. In the context of our interoperability model instantiation, the source modeling language corresponds to the DSML that represent the constructs of the MDD approach involved, and the target modeling language is the preexisting modeling language that will be customized with the specific MDD syntax. This identification of semantic equivalences can be performed by means of model mappings (weavings, or semantic links) between the constructs of the source metamodel and the target metamodel. Thus, these mappings guide the identification of the necessary extensions to integrate into the target metamodel the abstract syntax of the source metamodel. However, certain structural differences (heterogeneities) between the

involved metamodels may prevent the appropriate mapping specification, and, hence, they prevent the correct identification of the required metamodel extensions. This situation is presented in works such as [76] and [85]. These works propose systematic approaches for the generation of UML profiles starting from metamodel mappings. However, due to structural differences that are present in the involved metamodels, the final UML profile generation cannot be completely automated. These structural differences also affect the completeness of the obtained UML profile, which cannot customize the target modeling language with all the modeling information required.

Therefore, to solve this challenge, we propose the definition of a pivot metamodel that allows the structural differences to be fixed, and an appropriate mapping specification to be obtained. This pivot metamodel that we have called *Integration Metamodel* [28] provides the necessary information to perform the appropriate integration of modeling languages.

4.3 Third Challenge: Automatic Generation of Metamodel Extensions

Finally, the third challenge is related to how to automate the generation of the required metamodel extensions from the defined metamodels and the metamodel mappings. This is to automate the generation of the required UML profile. The automatic generation of the required metamodel extensions prevents the potential inconsistencies between the syntax of the source and target metamodel that a manual specification may produce. In addition, the effort in the implementation of the UML profiles is considerably reduced due to the automatic generation since it is not necessary to know specific details related to the correct UML profile specification or deal with complexity of large metamodels. The benefits obtained from the automatic UML profile generation are very relevant since, according to Selic in [76], the lack of knowledge about the features of the UML profile specification has produced that many of the existing UML profiles definitions be invalid or of poor quality.

In general terms, the metamodel extensions that must be implemented in the UML profile can be automatically identified by comparing the source and target metamodels according to the semantic equivalences identified (defined in the metamodel mappings). Thus, the extensions are the additional modeling information that is necessary to fix the differences that exist between the target and source metamodel. For instance, if in the source metamodel there is a property that cannot be mapped to the target metamodel, then the UML profile extends the target metamodel with this non-mapped property.

Thus, a UML profile can be automatically generated by considering all the possible metamodel differences, and, for each one of these, to define specific rules that generate the necessary UML profile extensions.

For the application of the proposed interoperability model a specific process has been defined. The stages and artifacts involved in this interoperability process

are defined by considering the solutions proposed to solve the three challenges presented. This interoperability process is detailed in the next section.

5 The Model-Driven Interoperability Process

In this section, we introduce the process to achieve and automate interoperability in model-driven developments, which is comprised by the following four steps: (1) Definition of modeling anguage metamodels; (2) Definition of integration metamodel; (3) Automatic UML profile generation; and (4) Generation of model-interchange mechanisms. Steps 2–4 of the process are based on original contributions that were created to tackle the interoperability challenges identified.

The modeling language integration is the core of the model-driven interoperability process proposed. It automates the generation of the necessary metamodel extensions and guides the specification of appropriate mappings, which are the main artifacts to perform the automatic model interchange. Thus, the first three steps of the interoperability process are related to perform the integration of the modeling languages involved. Figure 4 shows a BPMN [84] schema of the interoperability process proposed.

In the definition of this interoperability process, different works have been considered. Some of these works are: definition of UML profiles using DSML metamodels [27, 52, 76, 85], correct use of metamodels in software engineering [41], UML profile implementations,[1] interchange between UML profiles and DSMLs [3], and new UML profile features that are introduced in UML [65]. The four steps that comprise the interoperability process are detailed below:

Step 1: Definition of Modeling Language Metamodels. The first step of the process corresponds to the starting point proposed as solution of the first integration challenge presented in the previous section, which is the specification or selection of the EMOF metamodels of the involved modeling languages. As guidance to perform this step, the paper presented in [32] shows an interesting case study related to the UML association.

Step 2: Definition of Integration Metamodel. The second step is the definition of an Integration Metamodel to identify the equivalences between the metamodels involved and to fix the mapping issues that are produced by structural differences that may exist. Detailed example of how defining an Integration Metamodel and its application is presented in [28, 29].

Step 3: Automatic UML Profile Generation. This step considers the automatic generation of the UML profile that implements the metamodel extensions that are required to customize the abstract syntax of a target modeling language with the

[1]OMG: Catalog of UML Profile Specifications, http://www.omg.org/technology/documents/profile_catalog.htm

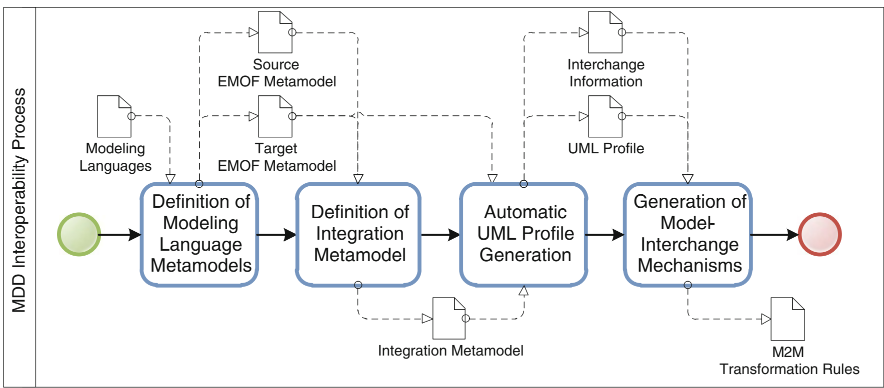


Fig. 4 Model-driven interoperability process

modeling information of the MDD approach involved. A suite of transformation rules and application example for this step has been presented in [30, 31]

Step 4: Generation of Model-Interchange Mechanisms. This step considers the generation of the necessary model transformations rules to automatically obtain from the models that are defined with the customized modeling language appropriate inputs (models) for specific modeling management tools, such as model compilers. The interchange mechanisms also transform source MDD models into equivalent representation using the customized modeling language. This is a bidirectional interchange of models. More details related to the development of this step can be found in [31].

For implementing an interoperability solution according to these four steps, existing open-source modeling tools can be used. For instance, the works presented in [32] and [34] implement a complete interoperability processes by considering the eclipse modeling tool facilities (UML profiles, eclipse metamodeling facilities (EMF), and ATL transformations). However, specific implementations can be also performed, for instance, in [29] we present a commercial tool that implements the model interchange mechanisms based on XSLT transformations.

Furthermore, the different artifacts that are involved in the application of the proposed interoperability process are defined to facilitate the validation and verification in each step. Some of the validation and verification facilities that can be obtained are the following:

- It is possible to verify the abstract syntax related to the modeling languages that must interoperate by means of the metamodels that are involved in the interoperability process. Also, the definition of metamodels by means of a standard metamodeling language (EMOF) facilitates the verification of the abstract syntax specified in relation to the supported semantics of the corresponding modeling languages.

- The construction of an Integration Metamodel facilitates the definition of specific rules for automatic UML profile generation. It allows the definition of verification mechanism that assures the correct application of these rules, and, hence, the correct generation of the resultant UML profile.
- The interchange of models is based on specific model-to-model transformation rules, which are based on the generated metamodel extensions and mappings. This allows the implementation of validation mechanisms to assure that the metamodel extensions and the defined models are defined according to the specification of the MDD approach involved.

6 Conclusions

In this chapter, we propose a conceptual model, which is used as reference to identify the necessary tools, and artifacts for the definition of concrete model-driven interoperability solutions.

The different elements considered in the proposed interoperability model have been instantiated by using the current model-based technologies. To complete the proposed interoperability model, we have defined a specific process, which indicates how the different elements of the proposed model can be coordinated to support the automatic interoperability in a model-driven context.

Thus, the interoperability process obtained is aligned with current modeling standards and technologies, such as modeling language specifications using metamodels, metamodel extension mechanisms that are implemented as UML profiles, and interchange mechanisms that are implemented through model transformations. Also, time and defects related to manual specification of metamodel extensions and transformation rules are reduced by means of the automatic generation of the interoperability artifacts involved.

The structure proposed for the interoperability process is also a suitable reference for other metamodel extension mechanisms or proposals for model interchange. This structure is easily adaptable to different model-based technologies. For instance, the UML profile generation rules can be changed to implement the required extensions with a different extension mechanism. The work presented by Bruck et al. in [15] introduces different approaches for the definition of metamodel extensions and provides a comparative summary about the approaches that are presented.

The adaptation to potential changes that the involved modeling languages may suffer is also improved by the proposed interoperability process. Changes in the modeling languages directly impact the defined metamodels. With the application of the proposed process, these changes are automatically propagated to the interoperability artifacts (metamodels extensions, and mappings). This is very important, especially when the involved modeling languages are comprised by a big number of conceptual constructs, which are permanently changing. In this context, the manual identification of the impact that a change in the modeling languages has over the

defined extensions and model transformation rules can be a titanic labor, which demands a lot of time and usually is error prone.

The report presented in [34] and the paper presented in [32] show two empirical evaluations of the proposal, which consider their industrial application in an industrial MDD scenario, and the definition of verification mechanisms to assure the completeness of the information exchanged.

Finally, the following works present the application of the interoperability model and process proposed in different contexts:

- Linking of goal-oriented modeling and MDD [5–7, 33, 68].
- Integration of domain-specific modeling languages and general-purpose modeling languages [29–32].
- Application of verification model in MDD contexts [56, 57].
- Business Modeling for Service Engineering [73, 74].
- Application of Software Maturity models in MDD processes [81].

References

1. A.W.P.: Levels of Information Systems Interoperability (LISI). Department of Defense, USA (1998)
2. Abdulhadi, S.: i* Guide Version 3.0 (August 2007). Available at http://istar.rwth-aachen.de/tiki-index.php?page=i%2A+Guide. Last access Apr 2013
3. Abouzahra, A., Bézivin, J., Fabro, M.D.D., Jouault, F.: A practical approach to bridging domain specific languages with UML profiles. Paper Presented at the Best Practices for Model Driven Software Development (OOPSLA'05), San Diego. ACM (2005)
4. Agostinho, C., Correia, F., Jardim-Goncalves, R.: Interoperability of complex business networks by language independent information models. Paper Presented at the 17th ISPE International Conference on Concurrent Engineering (CE 2010), Cracow. Springer (2011)
5. Alencar, F., Marín, B., Giachetti, G., Pastor, O., Castro, J., Pimentel, J.H.: From i* requirements models to conceptual models of a model driven development process. Paper Presented at the 2nd Working Conference on the Practice of Enterprise Modeling (PoEM 2009), Stockholm. Springer (2009)
6. Alencar, F., Marín, B., Giachetti, G., Pastor, O., Castro, J., Franch, X., Pimentel, J.: From i* to OO-method: problems and solutions. Paper Presented at the Fourth International i* Workshop (istar 2010), CAiSE Workshops, Hammamet. CEUR-WS (2010)
7. Alencar, F.M.R., Pastor, O., Marín, B., Giachetti, G., Castro, J.: Aligning goal-oriented requirements engineering and model-driven development. Paper Presented at the 11th International Conference on Enterprise Information Systems (ICEIS). Milan. Springer (2009)
8. Barnickel, N., Fluegge, M.: Towards a conceptual framework for semantic interoperability in service oriented architectures. Paper Presented at the 1st International Conference on Intelligent Semantic Web-Services and Applications, Amman. ACM (2010)
9. Baumgart, A.: A common meta-model for the interoperation of tools with heterogeneous data models. Paper Presented at the 3rd European Workshop on Model Driven Tool and Process Integration (MDTPI), Paris. Fraunhofer Institute for Open Communication Systems (2010)
10. Berger, S., Grossmann, G., Stumptner, M., Schrefl, M.: Metamodel-based information integration at industrial scale. Paper Presented at the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo. ACM/IEEE (2010)
11. Berre, A.-J., Liu, F., Xu, J., Elvesaeter, B.: Model driven service interoperability through use of semantic annotations. Paper Presented at the International Conference on Interoperability for Enterprise Software and Applications China (IESA '09), Beijing. Springer (2009)

12. Biehl, M., Sjöstedt, C.-J., Törngren, M.: A modular tool integration approach—experiences from two case studies. Paper Presented at the 3rd European Workshop on Model Driven Tool and Process Integration (MDTPI), Paris. Fraunhofer Institute for Open Communication Systems (2010)
13. Blanc, X., Gervais, M.-P., Sriplakich, P.: Model bus: towards the interoperability of modelling tools. Paper Presented at the Model-Driven Architecture: Foundations and Applications (MDAFA), Linköping. Springer (2004)
14. Brambilla, M., Fraternali, P., Tisi, A.M.: A transformation framework to bridge domain specific languages to MDA. Paper Presented at the Models in Software Engineering Workshops and Symposia at MODELS 2008, Toulouse. Springer (2008)
15. Bruck, J., Hussey, K.: Customizing UML: Which Technique is Right for You? IBM, (2007). Available at http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html. Last access April 2013
16. Bruneliere, H., Cabot, J., Clasen, C., Jouault, F., Bezivin, J.: Towards model driven tool interoperability: bridging eclipse and microsoft modeling tools. Paper Presented at the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010), Paris. Springer (2010)
17. Costa, R., Garcia, O., Nuñez, M., Maló, P., Gonçalves, R.: Integrated solution to support enterprise interoperability at the business process level on e-procurement. Paper Presented at the 3rd International Conference on Interoperability of Enterprise Software and Applications (I-ESA 2007), Funchal - Madeira Island. I-ESA (2007)
18. Coutinho, L., Brandao, A., Sichman, J., Boissier, O.: Model-driven integration of organizational models. Paper Presented at the IX Agent-Oriented Software Engineering (AOSE). Lecture Notes in Computer Science, vol. 5386, pp. 1–15 (2009)
19. Crnkovic, I., Malavolta, I., Muccini, H.: A model-driven engineering framework for component models interoperability. Paper Presented at the International Symposium on Component Based Software Engineering (CBSE), East Stroudsburg. Springer (2009)
20. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. Paper Presented at the First International Workshop on Model-Driven Interoperability (MDI), Oslo (2010)
21. Eclipse: Eclipse Modeling Framework Project. Available at http://www.eclipse.org/modeling/emf/. Last access Apr 2013
22. Eclipse: Eclipse Modeling Project. Available at http://www.eclipse.org/modeling/. Last access Apr 2013
23. Eclipse: Graphical Modeling Framework Project. Available at http://www.eclipse.org/modeling/gmp/. Last access Apr 2013
24. Eclipse: UML2 Project. Available at http://www.eclipse.org/uml2/. Last access Apr 2013
25. Fabro, M.D.D., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Softw. Syst. Model. **8**(3), 305–324 (2009)
26. Falbo, R.A., Guizzardi, G., Duarte, K.C.: An ontological approach to domain engineering. Paper Presented at the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ischia Island. ACM (2002)
27. Fuentes-Fernández, L., Vallecillo, A.: An introduction to UML profiles. Eur. J. Inform. Professional (UPGRADE) **5**(2), 5–13 (2004)
28. Giachetti, G., Valverde, F., Pastor, O.: Improving automatic UML2 profile generation for MDA industrial development. Paper Presented at the 4th International Workshop on Foundations and Practices of UML (FP-UML), ER Workshop, Barcelona. Springer (2008)
29. Giachetti, G., Marin, B., Pastor, O.: Integration of domain-specific modeling languages and UML through UML profile extension mechanism. Int. J. Comput. Sci. Appl. **6**(5), 145–174 (2009)
30. Giachetti, G., Marín, B., Pastor, O.: Using UML as a domain-specific modeling language: a proposal for automatic generation of UML profiles. Paper Presented at the 21st International Conference on Advanced Information Systems (CAiSE 2009), Amsterdam. Springer (2009)

31. Giachetti, G., Marín, B., Pastor, O.: Using UML profiles to interchange DSML and UML models. Paper Presented at the Third International Conference on Research Challenges in Information Science (RCIS), Fès. IEEE (2009)
32. Giachetti, G., Albert, M., Marín, B., Pastor, O.: Linking UML and MDD trough UML profiles: a practical approach based on the UML association. J. Universal Comput. Sci. (J.UCS) **16**(17) (2010)
33. Giachetti, G., Alencar, F., Marín, B., Pastor, O., Castro, J.: Beyond requirements: an approach to integrate i* and model-driven development. Paper Presented at the XIII Conferencia Iberoamericana en Software Engineering (CIbSE 2010), Cuenca. CIBSE (2010)
34. Giachetti, G., Alencar, F., Franch, X., Marín, B., Pastor, O.: Technical Report ProS-TR-2011-07: Automatic Verification of Requirement Models for Their Interoperability in Model-Driven Development Processes. Universidad Politécnica de Valencia, Spain (2011)
35. Giachetti, G., Marín, B., Valverde, F.: Interoperability for model-driven development—current state and future challenges. Paper Presented at the 6th International Conference on Research Challenges in Information Science (RCIS) (2012)
36. Guerra, E., Lara, J.D., Orejas, F.: Inter-modelling with patterns. Software. Syst. Model. (SoSym), **12**(1), 145–174 (2013)
37. Guha, R., Al-Dabass, D.: Impact of web 2.0 and cloud computing platform on software engineering. Paper Presented at the Electronic System Design (ISED), Bhubaneswar. IEEE (2010)
38. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "semantics"? IEEE. Comput. **37**(10), 64–72 (2004)
39. Haslhofer, B., Klas, W.: A survey of techniques for achieving metadata interoperability. ACM Comput. Surveys (CSUR) **42**(2) (2010)
40. Hein, C., Ritter, T., Wagner, M.: Model-driven tool integration with modelbus. Paper Presented at the 1st International Workshop on Future Trends of Model-Driven Development (FTMDD), Milan. SciTePress (2009)
41. Henderson-Sellers, B.: On the challenges of correctly using metamodels in software engineering. Paper Presented at the 6th Conference on Software Methodologies, Tools, and Techniques (SoMeT), Rome. IOS International (2007)
42. Höfferer, P.: Achieving business process model interoperability using metamodels and ontologies. Paper Presented at the 15th European Conference on Information Systems (ECIS 2007), St. Gallen. AIS (2007)
43. ICT: The future of cloud computing, opportunities for European cloud computing beyond 2010. In: Keith J., Burkhard N.-L. (eds.) EC ICT Research in FP7—Expert Group Report, ICT, European Commission. Information Society and Media (2010)
44. IEEE: IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries. IEEE, New York (1990)
45. Jankovic, M., Ivezic, N., Knothe, T., Marjanovic, Z., Snack, P.: A case study in enterprise modelling for interoperable cross-enterprise data exchange. Paper Presented at the 3rd International Conference on Interoperability of Enterprise Software and Applications (I-ESA 2007), Funchal - Madeira Island. I-ESA (2007)
46. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
47. Jouault, F., Guéguen, T.: Integration by model-driven virtual tool. Paper Presented at the 2nd European Workshop on Model Driven Tool and Process Integration (MDTPI) (2009)
48. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting metamodels to ontologies: a step to the semantic integration of modeling languages. Paper Presented at the MoDELS 2006, Genova. Springer (2006)
49. Kappel, G., Wimmer, M., Retschitzegger, W., Schwinger, W.: Leveraging model-based tool integration by conceptual modeling techniques. In: The Evolution of Conceptual Modeling. LNCS, vol. 6520, pp. 254–284. Springer, Berlin (2011)

50. Klar, F., Rose, S., Schürr, A.: A meta-model-driven tool integration development process. Paper Presented at the 2nd International United Information Systems Conference (UNISCON'2008), Klagenfurt. Springer (2008)
51. Kolovos, D., Paige, R., Rose, L., Polack, F.: The Epsilon Book. Eclipse Foundation (2010). Available at http://www.eclipse.org/epsilon/doc/book/. Last access April 2013
52. Lagarde, F., Espinoza, H., Terrier, F., Gérard, S.: Improving UML profile design practices by leveraging conceptual domain models. Paper Presented at the 22th IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta. IEEE/ACM (2007)
53. Loniewski, G., Insfran, E., Abrahao, S.: A systematic review of the use of requirement engineering techniques in model-driven development. Paper Presented at the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010) (2010)
54. Lukácsy, G., Szeredi, P., Benkő, T.: Towards automatic semantic integration. Paper Presented at the 3rd International Conference on Interoperability of Enterprise Software and Applications (I-ESA), Funchal - Madeira Island. I-ESA (2007)
55. Mahé, V., Brunelière, H., Jouault, F., Bézivin, J., Talpin, J.-P.: Model-driven interoperability of dependencies visualizations. Paper Presented at the 3rd European Workshop on Model Driven Tool and Process Integration (MDTPI), Paris. Fraunhofer Institute for Open Communication Systems (2010)
56. Marín, B., Giachetti, G., Pastor, O., Vos, T.E.J.: A tool for automatic defect detection in models used in model-driven engineering. Paper Presented at the 7th International Conference on the Quality of Information and Communications Technology (QUATIC), Porto. IEEE (2010)
57. Marín, B., Vos, T., Giachetti, G., Baars, A., Tonella, P.: Towards testing future web applications. Paper Presented at the 5th IEEE International Conference on Research Challenges in Information Science (RCIS 2011), Gosier. IEEE (2011)
58. Milanovic, N., Cartsburg, M., Kutsche, R., Widiker, J., Kschonsak, F.: Model-based interoperability of heterogeneous information systems: an industrial case study. Presented at the 5th European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA). Lecture Notes in Computer Science, vol. 5562, pp. 325–336 (2009)
59. Moreno, N., Vallecillo, A.: Towards interoperable web engineering methods. J. Am. Soc. Inf. Sci. Technol. **59**, 1073–1092 (2008)
60. Ohren, O.P., Chen, D., Grangel, R., Jaekel, F.-W., Karlsen, D., Knothe, T., Rolfsen, R.K.: ATHENA-A1, Deliverable DA1.5.2: Report on Methodology description and guidelines definition. Oslo (2005)
61. OMG: XMI 2.4.1 Specification (2011)
62. OMG: MOF 2.4.1 Core Specification (2011)
63. OMG: QVT 1.1 Specification (2011)
64. OMG: UML 2.4.1 Superstructure Specification (2011)
65. OMG: UML 2.4.1 Infrastructure Specification (2011)
66. Opdahl.: Incorporating UML class and activity constructs into UEML. Paper Presented at the International Conference on Advances in Conceptual Modeling: Applications and Challenges—ER 2010 Workshops, Vancouver. Springer (2010)
67. Ouksel, A.M., Sheth, A.: Semantic interoperability in global information systems. ACM SIGMOD **28**(1), 5–12 (1999)
68. Pastor, O., Giachetti, G.: Linking goal-oriented requirements and model-driven development. In: Nurcan, S., Salinesi, C., Souveyet, C., Ralyté, J. (eds.) Intentional Perpectives on Information Systems Engineering, pp. 257–276. Springer, Heidelberg (2010)
69. Polgár, B., Ráth, I., Szatmári, Z., Horváth, Á., Majzik, I.: Model-based integration, execution and certification of development tool-chains. Paper Presented at the Second European Workshop on Model Driven Tool and Process Integration (MDTPI), Enschede. Fraunhofer Verlag (2009)
70. Radjenovic, A., Paige, R.F.: Behavioural interoperability to support model-driven systems integration. Paper Presented at the 1st Workshop on Model Driven Interoperability (MDI 2010), Oslo. ACM (2010)

71. Roser, S., Bauer, B.: Improving interoperability in collaborative modelling. Paper Presented at the 3rd International Conference on Interoperability of Enterprise Software and Applications (I-ESA 2007), Funchal - Madeira Island. I-ESA (2007)
72. Sarantis, D., Charalabidis, Y., Psarras, J.: Towards standardising interoperability levels for information systems of public administrations. In: Yannis C., Hervé P., Euripidis L., Kai M. (eds.) eJETA Special Issue on "Interoperability for Enterprises and Administrations Worldwide". eJETA J. (2008)
73. Scheithauer, G., Kett, H., Kaiser, J., Hackner, S., Hu, H., Wirtz, G.: Business modeling for service engineering: a case study in the IT outsourcing domain. Paper Presented at the Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre (2010)
74. Scheithauer, G., Wirtz, G.: Business modeling for service descriptions: a meta model and a UML profile. Paper Presented at the 7th Asia-Pacific Conference on Conceptual Modelling (APCCM 2010), Brisbane, Australia. Australian Computer Society (2010)
75. Seifert, M., Wende, C., Aßmann, U.: Anticipating unanticipated tool interoperability using role models. Paper Presented at the 1st Workshop on Model Driven Interoperability, Oslo. ACM (2010)
76. Selic, B.: A systematic approach to domain-specific language design using UML. Paper Presented at the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), Santorini Island. IEEE (2007)
77. Staron, M., Wohlin, C.: An industrial case study on the choice between language customization mechanisms. Paper Presented at the Product-Focused Software Process Improvement (PROFES), Amsterdam. Springer (2006)
78. Sunindyo, W.D., Moser, T., Winkler, D., Biffl, S.: A process model discovery approach for enabling model interoperability in signal engineering. Paper Presented at the 1st Workshop on Model Driven Interoperability, Oslo. ACM (2010)
79. Tran, H., Zdun, U., Dustdar, S.: View-based reverse engineering approach for enhancing model interoperability and reusability in process-driven SOAs. Paper Presented at the 10th International Conference on Software Reuse (ICSR 2008), Beijing. Springer (2008)
80. Vallecillo, A.: On the combination of domain specific modeling languages. Paper Presented at the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010), Paris. Springer (2010)
81. Vasconcelos, A.M.L., Giachetti, G., Marín, B., Pastor, O.: Towards a CMMI-compliant goal-oriented software process through model-driven development. Paper Presented at the Practice of Enterprise Modeling (POEM), Oslo. Springer (2011)
82. W3C: XML Web Page. http://www.w3.org/XML/. Accessed April 2013
83. Wang, W., Tolk, A., Wang, W.: The levels of conceptual interoperability model: applying systems engineering principles to M&S. Paper Presented at the 2009 Spring Simulation Multiconference (SpringSim'09), San Diego. ACM (2009)
84. White, S.A., Miers, D.: BPMN Modeling and Reference Guide. Future Strategies Inc., Lighthouse Pt (2008)
85. Wimmer, M., Schauerhuber, A., Strommer, M., Schwinger, W., Kappel, G.: A semi-automatic approach for bridging DSLs with UML. Paper Presented at the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM), Montréal. University of Jyväskylä, Jyväskylä (2007)
86. Ziemann, J., Ohren, O., Jäkel, F.-W., Kahl, T., Knothe, T.: Achieving enterprise model interoperability applying a common enterprise metamodel. Paper Presented at the 2nd International Conference on Interoperability of Enterprise Software and Applications (I-ESA 2006), Bordeaux (2007)

Domain and Model Driven Geographic Database Design

Jugurta Lisboa-Filho, Filipe Ribeiro Nalon, Douglas Alves Peixoto, Gustavo Breder Sampaio, and Karla Albuquerque de Vasconcelos Borges

Abstract After many years of research in the field of conceptual modeling of geographic databases for Geographic Information Systems, experts have produced many different alternatives of conceptual data models from extensions of the Entity-Relationship model or of Unified Modeling Language (UML). However, the lack of consensus on which is the most suitable one for modeling applications in the geographical domain brings up a number of problems for field advancement, mainly problems of interoperability of database design and CASE tools. The Model Driven Architecture (MDA) approach allows the development of systems from an abstract view until the corresponding implementation code that can be automatized by means of models transformation. A UML Profile is an extension mechanism of UML which allows a structured and precise extension of its constructors, being a good solution to standardize domain-specific modeling, as it uses the entire UML infrastructure. This chapter describes the use of MDA approach in the design of databases in geographical domain; using a UML Profile called GeoProfile aligned with international standards of ISO 191xx series. The chapter also shows that with the automatic transformation of models it is possible to achieve the generation of scripts for spatial databases from a conceptual data schema in a high level of abstraction.

Keywords Conceptual data modeling • Geographic database • Model driven architecture • UML Profile

J. Lisboa-Filho (✉) · D.A. Peixoto · G.B. Sampaio
Universidade Federal de Viçosa, Departamento de Informática
e-mail: jugurta@ufv.br; douglasalves.ufv@gmail.com; gustavobreder@gmail.com

F.R. Nalon · K.A. de Vasconcelos Borges
Prodabel - Empresa de Informática e Informação do Município de Belo Horizonte
e-mail: frnalon@gmail.com; karla@pbh.gov.br

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3_15, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

The activity of software development is a task that requires increasing use of standardized methodologies and techniques that are widely known. Currently, the main concern of the designer is a good understanding of the problem domain in order to generate solutions that suit the real necessities of the users.

In order to assist in this task of understanding the problem and reducing the system complexity to be developed, the main technique that is used is modeling. A model is a reality simplification [9]. In database design, the construction of models in steps helps to design the database structure without having to worry about implementation details.

In the last 20 years, research has aimed to create or adapt conceptual data models for geographic applications. The existence of several models has brought a problem to the area, which is the lack of a modeling standard. Tools have been created for different models and it is difficult to obtain interoperability among the created solutions.

For the standardization of these models, a Unified Modeling Language (UML) profile called GeoProfile [21] was proposed. A profile is an extension mechanism of the UML, which allows customizing the UML to a specific domain. The GeoProfile was proposed for conceptual data modeling in the geographic domain, which puts together the characteristics of the main existing spatial data conceptual models. The construction process of the GeoProfile can be compared with a step of the domain engineering, which is the domain analysis, in which the domain knowledge is studied and analyzed. According to Falbo et al. [10], one of the domain analysis goals is to make possible the reuse of the domain model generated for a group of applications. One of the UML profile construction outcomes is a domain metamodel. This metamodel contains reusable requirements, of the intended domain, to build applications in that domain.

Furthermore, as an effort for the geographic information standardization, some organizations, such as the International Organization for Standardization (ISO) and the Open Geospatial Consortium (OGC), have published international standards to help in the construction of standardized geographic applications.

The objective of this chapter is to describe the use of the Model Driven Architecture (MDA) approach in the design of databases in geographical domain, using the GeoProfile aligned with international standards of ISO 191xx series. The chapter also shows that with the automatic transformation of models it is possible to achieve the generation of scripts for spatial databases from a conceptual data schema in a high level of abstraction.

Section 2 presents the main concepts related to modeling of geographic databases, describes the main international standards for geographic information, and summarizes the GeoProfile. Section 3 describes the steps and types of models used in the MDA approach. Section 4 describes the process of designing geographic databases based on the MDA approach. Section 5 illustrates the entire process of designing a geographic database, based on a case study of a system in the field of sugarcane crop for ethanol production. Some conclusions are presented in Sect. 6.

2 Geographic Database Modeling

2.1 Basic Requirements

One of the main components of a Geographic Information System (GIS) is the storage component denominated geographic database, whose function is to structure and store the data in order to allow analysis operations involving spatial and alphanumeric data [35].

Due to the complexity of GIS applications, a major challenge in developing these systems has been designing the database, since this type of project requires the use of different tools, as the activities required for their preparation vary according to the complexity of the system, the type of personnel involved, the database management system (DBMS) used, etc.

The database design is traditionally done in three stages: conceptual, logical, and physical [8]. According to Borges et al. [4], for applications in the geographical domain, the level of conceptual representation provides a set of concepts with which the geographic phenomena, such as rivers, buildings, roads, and vegetation, can be modeled at a high level of abstraction, as perceived by the user. Classes to be created in the database are defined at this level, which are possibly associated with some kind of spatial representation.

Parent et al. [27] highlight some advantages of using conceptual modeling in applications that manipulate geospatial data. First, users can express their knowledge of the system using concepts that are close to their reality and independent of computing concepts. Moreover, as conceptual modeling is independent of system implementation, the result of modeling remains valid in case of technological changes. That is, the developed scheme can be reused regardless of the GIS chosen for the system implementation. Finally, due to its readability, conceptual modeling promotes the exchange of semantic information referring to the project.

Because of the particularities of the geographic information, several specific solutions for the modeling of geographic data have emerged in recent years. Lisboa-Filho and Iochpe [19] proposed a list of key requirements for modeling geographic data as follows:

- Geographic phenomena and conventional objects: In a geographic database, in addition to the data on geospatial phenomena, there are generally conventional data, such as those contained in any information system. A farm, for example, can be a geographical phenomenon if its spatial information is stored in the database, such as its boundaries. On the other hand, the information about the owner of the farm can be an example of a conventional object if it does not present spatial features.
- Field and object views: The classification of geographic phenomena in these two views is intended to represent properly the geographical reality observed. While in the object view the real world consists of entities and individual well-defined spatial boundaries (e.g., rivers, plots, and streets), in the field view, the real world

is understood as a set of attributes that vary continuously in space (e.g., relief, soil type, temperature).

- Spatial aspects: This requirement refers to the need of connecting fields and geographic objects to an abstract spatial form for their representation. In the object view the phenomena are represented by points, lines, polygons, or their combinations. In the field view, a continuous surface can be represented by numerical models, sets of isolines or grid of cells, for example. The type of representation to be used depends on the purpose of the application, the representation scale, and the shape of the phenomenon.
- Thematic aspects: In a GIS, geographic entities are not treated in isolation. They are grouped according to the characteristics and relationships they have in common. The division of a system by themes (e.g., hydrography, vegetation, urban planning) allows modeling simplification and facilitates understanding of the area by the designer.
- Multiple representations: Users can have different views of the same phenomenon, which are probably represented in different scales or projections. For example, a city can be represented as a point or a polygon depending of the data scale.
- Spatial relationships: The identification of the types of relationships that must be kept in the database is a complex problem, since the number of possible relationships is large in the geographical area, because of the spatial interactions that may occur among the phenomena. For example, a road can cross a river, but this relationship can be or not be held in the database.
- Temporal aspects: The storage of the changes that occur in geographic features is important for a better understanding of the phenomena and making predictions. For example, the limits of a parcel can change over the years.

Other similar classifications were also proposed, for example, Friis-Christensen et al. [11] divide the requirements for modeling geographic data into five groups as follows: spatial-temporal properties, roles, associations, constraints and data quality. In a deeper analysis, it is possible to confirm that both classifications are equivalent in many aspects.

Pinet [28] lists a series of conceptual models of specific data for the geographical domain. Among them we can highlight some models based on objects, such as GeoOOA [17], OMT-G [4], MADS [27], UML-GeoFrame [20] and the PVL model of the Perceptory tool [1]. Each model has particular characteristics and seeks to meet the requirements for geographic application modeling. Based on these models and in accordance with International Standards for Geographic Information shown in the next section, the UML GeoProfile was proposed as described in the Sect. 2.3.

2.2 International Standards for Geographic Information

Standards are used in many fields of society and some of them are better known as, for example, the series of standards ISO 9000, which is quite common in

organizations, because it defines a set of rules and guidelines for quality management in an organization. With respect to geographic information, some efforts to standardize work have been undertaken by organizations like the ISO and the OGC.

These organizations are examples of the two types of groups that exist for international standardization, which are the international organizations and international consortia. International organizations base their decisions on consensus and are independent of the interests of individual industries or governments. Their standards are known as *de jure* standards. An example of international organization is the ISO. On the other hand, the international consortia are made up primarily of members from industry, government agencies, and universities. The standards developed by consortia are called industry standards or *de facto* standards. The OGC can be considered the most important consortium in the geographic information community [18]. According to Brodeur and Badard [5], the development of standards for geographic information aims to reduce the inconsistency between *de jure* and *de facto* standards.

2.2.1 The ISO 191xx Series of the ISO/TC 211 Technical Committee

The Technical Committee ISO/TC 211 is the one responsible for the preparation of the ISO 191xx series, which defines the international standards regarding the geographic information field. These standards aim to promote the usage of geographic information in an efficient, effective, and economical way, thus contributing to the solution of global problems, such as the humanitarian and ecological problems [34].

The ISO 191xx series standards are divided into specific groups. As listed in the ISO/TC 211 [34], there is the group of standards that specify the infrastructure for the geospatial standardization, standards that describes data models for geographic information, standards for geographic information management, standards for geographic information services, standards for encoding of geographic information, and standards for specific thematic areas. These standards can contribute in several levels of abstraction, since modeling up to the consideration of implementation aspects. In this chapter some standards related to data models for geographic information, more specifically the ISO 19107 Spatial Schema [32], ISO 19108 Temporal Schema [31] and ISO 19123 Schema for Coverage Geometry and Functions [33] standards, are analyzed.

The ISO 19107 Spatial Schema standard specifies a schema to describe and manipulate the spatial characteristics of the geographic features. A feature is an abstraction of a real world phenomenon. This abstraction is a geographic feature if it is associated with a relative localization on the Earth [32]. The standard consists of classes' diagrams that can be used in application schema, profiles, and implementation specifications. It also defines spatial operations, standards for use in the access, query, management, processing, and data exchange of geographic objects. The ISO 19107 standard defines in detail the geometric and topological characteristics that are necessary to describe the geographic features.

The ISO 19108 Temporal Schema standard defines the concepts regarding the temporal characteristics of geographic information, showing how these characteristics are abstracted from the real world. Jensen [14] considers two types of time: the valid time and the transaction time. The first one is the time when a fact is true in the observed reality and it is generated by the user. The second one is the time when a fact is stored in a database and it can be recovered. This international standard emphasizes the valid time instead of the transaction time. The standard consists of a class hierarchy that considers the geometric and topological aspects of the temporal characteristics [31].

The ISO 19123 Schema for Coverage Geometry and Function standard, on the other hand, defines a schema for the spatial characteristics of coverage. Coverage is a feature that has multiple values for each type of attribute and can represent a simple feature or a set of features. They integrate discrete and continuous geographic phenomena [33]. Examples of coverage include raster, TIN, point coverage, and polygon coverage. They are used in several specific areas such as, remote sensing, meteorology, soils, and vegetation.

Some related works have analyzed conceptual data models and their integration with geographical standards. Belussi et al. [3] describe the conceptual data model GEOUML in which a geographic database schema can be designed from the specialization of ISO TC211 standards. However, this model does not use graphic symbols for representing phenomena's spatial representation, which is a feature presented in various models proposed in the literature [1]. A study where the model elements of the Perceptory tool are related to the ISO standards is presented in [6].

2.2.2 Open Geospatial Consortium Standards

The OGC is currently the main consortium responsible for developing industry standards for Geographic Information Systems. Its development process is different from the ISO approach. The OGC develops specifications mainly focused on implementation, while ISO develops more abstract specifications.

Despite these minor differences, both ISO and OGC have developed cooperative agreements to harmonize their work and to develop future work [6]. For example, the document OpenGIS—Simple Feature Access is also recognized by the ISO under the name ISO 19125. The document is divided into two parts. The first (Common Architecture) describes a common architecture for simple geographical features and the second part (SQL option) describes an SQL implementation of the model described in the first part. Both parts deal with simple features, namely features whose geometry is restricted to two dimensions. The OGC standard used is the Simple Feature Access, as well as GeoProfile and projects using the MDA approach, both described in this chapter; this will be handled as its corresponding ISO 19125.

2.3 GeoProfile: A UML Profile for GIS Databases

The UML is a visual modeling language used for documenting artifacts of software systems. Despite being a general purpose language that can be used in various application domains, there are situations in which the elements of the UML are not able to effectively express some concepts of particular domains. Thus, the language provides extension mechanisms that allow customizing it to suit a specific application domain as follows:

- Stereotypes: A stereotype defines how an existing metaclass may be extended and enables the use of specific terminology for a domain or different platform in place of or in addition to the terminology used for the extended metaclass. Stereotypes can also change the appearance of the elements of the extended model using graphic icons.
- Tagged values: They are additional meta-attributes associated with a metaclass of the metamodel extended by a profile and add information to elements of the model.
- Constraints: These are restrictions associated with the corresponding elements of the metamodel. They can be written using natural languages or in Object Constraint Language (OCL), which is also standardized by the Object Management Group (OMG).
- Profile: A UML profile is a set of extension mechanisms grouped in an UML package stereotyped as profile.

A well-specified UML profile will have direct support of Computer Aided Software Engineering (CASE) tools. In other words, once the profile is defined, there is no need to implement new CASE tools. Enterprise Architect [30] and Rational Software Modeler [13] are examples of CASE tools with support for UML profiles.

The UML profile denominated GeoProfile [21, 23] was proposed to integrate the features of the major geographic data conceptual models. Thus, the GeoProfile is not a new model, but rather a compilation and integration from the builders of specific GIS applications present in the main models in the literature. With this, it is possible to use all the elements and advantages of UML 2.0. In addition, a designer familiarized with a particular model can customize GeoProfile, making it look like that model. The GeoProfile comprises a metamodel and a set of stereotypes, described in the following subsections.

2.3.1 The GeoProfile Metamodel

The GeoProfile metamodel, as defined in [21], is showed in Fig. 1. A geographic database comprises a number of themes, each of presented as a `Theme` metaclass. A theme can be formed by the aggregation of other themes or objects with or without spatial representation, characterized by the classes `GeoPhenomenon` and `ConventionalObj`, respectively.

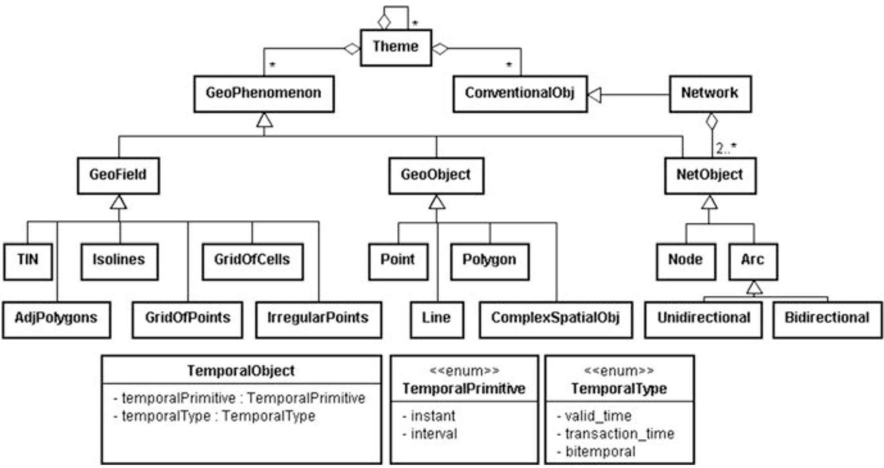


Fig. 1 Metamodel for the geographical domain

When one chooses to associate a spatial representation with objects of a class, it is possible that the phenomenon is perceived in the geographic field view (`GeoField`) or object view (`GeoObject`). Depending on the technique used in geographic information acquisition in the field, its representation is selected from six options as described in [12]: `AdjPolygons`, `Isolines`, `TIN`, `GridOfPoints`, `GridOfCells` or `IrregularPoints`. Representation of geographic objects can be of the types `Point`, `Line`, `Polygon` or `ComplexSpatialObj` (the object geometry consists of other geometries). To specify multiple representations, it is possible to use more than one stereotype in the same class of the conceptual schema.

The metaclass `Network` is used to modeling a whole network structure and contains only alphanumeric attributes which describes the general features of the network. Since this metaclass does not have spatial information, it was defined as a specialization of `ConventionalObj`. The networks are formed by `NetObject` objects, which can be nodes (`Node`), unidirectional arcs (`Unidirectional`), or bidirectional arcs (`Bidirectional`).

GeoProfile also indicates whether a class is considered temporary or not. In this case, it is implied that both the attributes and spatial data of an object can vary, and these changes must be maintained in the database. In this way, the metaclass `TemporalObject` was added to the metamodel. This metaclass has two attributes that characterize temporal information. One of these attributes indicates the temporal type (validity time, transaction time or bitemporal time), whereas the other defines the used temporal primitive type (instant or interval). There are two enumerations (`TemporalType` and `TemporalPrimitive`) for the possible values these attributes can assume.

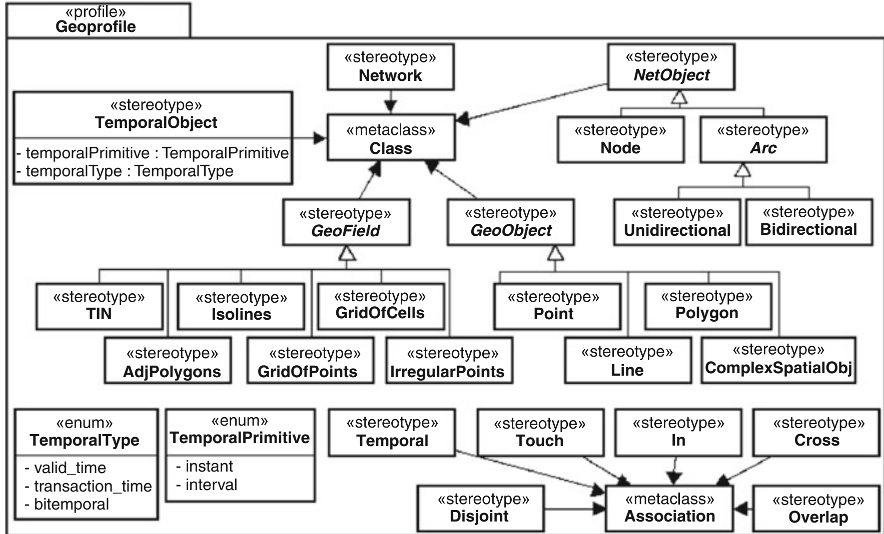


Fig. 2 GeoProfile's stereotypes

2.3.2 GeoProfile's Stereotypes

After creating the domain metamodel, the next step is to extend the UML metaclasses to create the profile itself. Figure 2 illustrates the stereotypes that have been defined for GeoProfile, which extend the metaclasses `Class` and `Association` of the UML. The black arrows refer to an extension relation between a stereotype and an UML element. This is the mechanism proposed by OMG to extend the UML in the following way: a basic UML element can be used to represent an element of a generic domain; in this case the UML elements `Class` and `Relationship` are used to represent the GeoProfile elements.

The white arrows refer to a specialization relation between UML stereotypes. This kind of relation is called *Is_A*, where a stereotype *subclass* inherits all the features and information from another stereotype *superclass*. New information and features can be added to the stereotype *subclass*, differentiating it from the others. The stereotypes `Node` and `Arc`, for example, are specializations of the stereotype `NetObject`. Both stereotypes, `Node` and `Arc`, are subclasses of `NetObject` and have common features inherited from `NetObject`, but each one of them can hold its own distinct features and information.

The `GeoObject` and `GeoField` stereotypes represent the geographic phenomena perceived in the objects and fields views, respectively. Since these stereotypes were defined as abstracts, as well as the `NetworkObj` and `Arc` stereotypes, they will not be included in the data schema during the modeling using the GeoProfile, but their corresponding subclasses will.

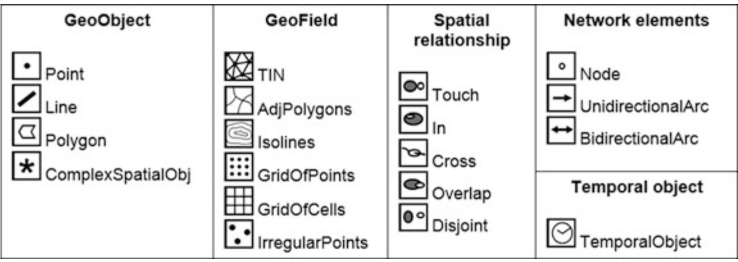


Fig. 3 Graphical notation for stereotypes

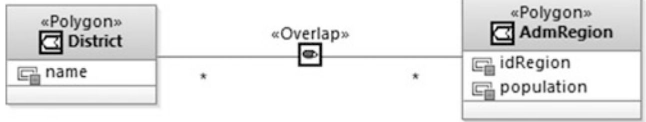


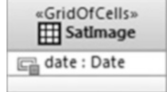
Fig. 4 An example of a conceptual schema using GeoProfile

To deal with temporal aspects, the `TemporalObject` stereotype, that also extends the metaclass `Class`, was included. The two enumerations that were included (`TemporalPrimitive` and `TemporalType`) are used to list the possible values that the meta-attributes (tagged values) `temporalPrimitive` and `temporalType` may assume, which are: `instant` and `interval`.

Besides the extensions to the metaclass `Class`, extensions to the metaclass `Association` were included. These extensions are aimed to creating stereotypes to serve the topological relationships [7], which are: `Touch`, `In`, `Cross`, `Overlap`, and `Disjoint`. In addition, designers are allowed to indicate that an association between two objects is only valid for one period and this history should be kept in the database. This is done by simply assigning the stereotype `Temporal`.

An important requirement for the project of UML profiles described in [26] is the definition of a graphical notation for the stereotypes. In the modeling of geographic databases, the use of this feature to represent the spatial characteristics of geographic objects is used in many models, for example, pictograms, initially developed by Bédard and Paquette [2], which influenced many models that emerged later. They help improve clarity and make the modeling more intuitive for the designer and easily understood by users. Figure 3 shows a set of icons that can be added to GeoProfile stereotypes. These icons were also based on the models mentioned above (UML-GeoFrame, OMT-G, MADS, and GeoOOA Perceptory's model), but designers accustomed to using a particular model can customize these icons as they wish.

Figures 4 and 5 illustrate some examples of classes modeled with GeoProfile stereotypes in graphic and textual forms. Figure 4 illustrates an example of spatial relationship between two classes (`District` and `AdmRegion`) with polygon spatial representations, specified by the stereotype `Polygon`. The stereotype



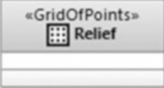


Fig. 5 Examples of classes using stereotypes for the field view

`Overlap` shows the spatial relationship that occurs between the classes. Figure 5 illustrates three examples of classes with spatial representation in the field view. The `SatImage` class with the stereotype `GridOfCells`, the `Humidity` class with the stereotype `IrregularPoints` and the `Relief` class with the stereotype `GridOfPoints`.

Besides the stereotypes, some constraints were also added, which are useful for the conceptual schema validation. Those constraints basically prevent the occurrence of three error types: addition of incompatible stereotypes with a same element, poor network construction, and addition of impossible topological relationships between two elements (e.g., cross relationship between two geographic objects with point representation). These three constraints groups were analyzed and a set of OCL expressions was specified. OCL has been frequently used to specify additional integrity constraints to the UML diagrams [29].

The code below shows one of the GeoProfile's OCL constraints, which is applied to the stereotype `GeoField`. This constraint defines that each class stereotyped as a geofield (`context GeoField`) must capture all stereotypes applied to this class (`getAppliedStereotypes`). If the output is stereotyped as a geographic object (`Point`, `Line`, `Polygon`, or `ComplexSpatialObj`) through the method `select`, the result set must be empty (`isEmpty`). This constraint checks for incompatible stereotypes in a class which has been assigned the stereotype `GeoField` in the schema. More details regarding all OCL constraints and how to implement them in a UML profile can be accessed in the GeoProfile's Web page at www.dpi.ufv.br/projetos/geoprofile.

```
context GeoField
inv: self.getAppliedStereotypes() ->
select(s | s.name = 'Point' or s.name = 'Line' or
 s.name = 'Polygon' or s.name = 'ComplexSpatialObj')
 -> isEmpty()
```

Finally, although the GeoProfile is most commonly used to static data schema designs, the class behavioral modeling, that is, the specification of the applicable operations to instances of a class, can be done naturally within this class, using methods specified in the UML.

3 Model-Driven Architecture

To improve software development OMG has adopted the MDA approach, which emphasizes the use of models. In this approach, the software development process is directed by the modeling activity of the system. A system model is a description using a specific notation. The artifacts produced in MDA are formal models, that is, models that can be understood by computers [25].

In MDA, the system requirements are modeled using a Computation Independent Model (CIM). This model is called domain model or business model and it uses a familiar vocabulary to the domain experts. A CIM does not show details of the systems structure, but of the environment in which the system will operate. This kind of model provides a useful way to understand the problem itself [9, 25].

In the second level of abstraction we find the Platform Independent Model (PIM). This is a model with an abstraction level relatively high and independent from any implementation technology [9, 16, 25].

Later, the PIM is transformed into a Platform Specific Model (PSM). A PSM is customized in order to specify the system in terms of implementation constructors which are available in a specific implementation technology. For instance, a PSM relational database include terms such as `table`, `column`, `foreign key`, among others. A PIM can be transformed into one or more PSMs. For each specific technology platform, a separate PSM is generated. The following step is the transformation of each PSM to source code. This transformation is relatively direct since the PSM is adjusted to the selected technology. Figure 6 illustrates the different levels of abstraction of MDA approach, showing the CIM as the highest level of abstraction model and the others, PIM and PSM, as inferior levels.

The CIM, PIM, and PSM are shown as artifacts in different steps in the system development life cycle, and they represent different abstraction levels in its specification as well. The ability of transforming a high level CIM into a PIM and later, transforming a PIM into a PSM increases the abstraction level in which a designer can work. This allows a designer to face more complex systems with fewer struggles [9, 25].

The development process using MDA approach may be compared with the process of domain engineering, which highlights three main steps: domain analysis, domain design, and domain implementation [10]. At the domain analysis step, the domain requirements are defined. These requirements are expected to be reusable, such as the CIM in the MDA approach, which is used to understand the problem itself. In the domain design step a generic and independent of platform architecture is established, such as the PIM level. Finally, at the domain implementation step the identification of reusable assets is done, as well as the architecture and components implementation, such as in the PSM level of the MDA, which transforms the considered PIM into a specific platform.

One of the aims of the MDA approach is to reduce the system development time. For this purpose, models in different abstraction levels are used, starting with models in high abstraction levels. Therefore, one of the challenges is transform high level

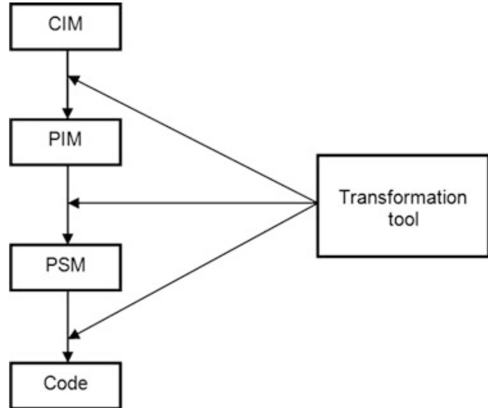


Fig. 6 Levels of abstraction of the MDA approach

models into lower level models. The transformation of models is the process of converting a model into another model that represent the same system [25].

An important characteristic of the MDA is that the transformations are automatically executed. Traditionally, the transformations from model to model or from model to code are manual. In the MDA approach, on the other hand, transformations are executed preferably by tools [16].

An automatic mapping is specified using a language to describe the transformation of a model into another. A desirable quality of a transformation language is portability; this enables the use of a mapping with different tools [25].

Some tools, available in the market, for supporting the MDA approach, have mechanisms for transforming predefined templates, but the ideal is to offer support to a language that enables users to customize the transformation of models as needed. An example of such language is the Atlas Transformation Language (ATL) developed by the research group ATLAS INRIA & LINA [24].

This language allows the definition of transformation rules, in which, given a schema created in a model of entry, along with transformation rules, generates a new schema in the output model, according to those rules. Thus, this language allows the transformation of a schema made from GeoProfile, for example, into another specific conceptual model, allowing the exchange of information between models and giving the designer flexibility in creating a schema. This approach can also be used in the transformation of a schema in each of the three levels of MDA; in this case, using the ATL language, it is necessary to define the transformation rules for each level.

4 Modeling Geographic Databases Using MDA

The use of MDA is not specific to the geographical domain, but it was used in this work to exemplify a domain engineering on the geographical field of study.

The development of GeoProfile was mainly motivated by the fact that UML can be used, along with all its available resources, for example, CASE tools, to model a geographic database.

The development of GeoProfile based on international standards is in accordance with the abstraction levels of the MDA approach. A major benefit of this approach is the productivity gain through the emphasis on modeling and the transformation of high-level models to lower-level models in an automated way [16]. Thus, the design of geographic databases can also take advantage of these benefits. For example, using tools that support the transformations will make it possible to generate, from the GeoProfile, lower-level models and, later, the database scripts for specific technologies, such as Oracle Spatial and PostGIS.

In related studies, Miralles and Libourel [22] propose a framework to design and implement spatial-temporal databases following the MDA approach, but this framework does not consider the CIM level, but suggests the use of the Perceptory tool to specify the business model. Bérdard and Larrivée [1] also mention the MDA approach as a key application for the use of the Perceptory model and consider the three levels of abstraction, namely, CIM, PIM, and PSM.

4.1 CIM Level

At this level of abstraction, only aspects related to the problem's domain are addressed, without dealing with implementation details. For the conceptual model of the database, the GeoProfile is used at this level, because it is designed to help designers in the first steps of a database project. The concern is to represent *which* are the spatial features of a particular geographic element and not *how* these features will be implemented. The use of stereotypes helps in this direction, since they make the model more intuitive for the user to understand the spatial features that are being represented.

Figure 7 illustrates an example of a schema modeled with GeoProfile at this level of abstraction. The schema shows four classes, three of them with spatial features and thus are stereotyped considering the notation proposed in Fig. 3.

4.2 PIM Level

After constructing the initial model of the database using GeoProfile, this model is transformed into a PIM model. At this level of abstraction, the elements of international standards are taken into account. To make the transformation, the GeoProfile stereotypes were mapped to specific classes of international standards. Table 1 illustrates the mapping carried out with the standards ISO 19107, ISO 19108, and ISO 19123. However, similar mapping can also be made for OGC standards. Due to lack of space, these will not be seen in this chapter.

Fig. 7 Example of a conceptual schema at the CIM level of abstraction

Table 1 Correspondence between the GeoProfile elements and the ISO 191xx standards

Requirements of GeoDB modeling	GeoProfile	Classes in the ISO standards	Standard
Geographical objects in the object view	Point	GM_Point	ISO 19107
	Line	GM_Curve	ISO 19107
	Polygon	GM_Surface	ISO 19107
	ComplexSpatialObj	GM_Complex	ISO 19107
Geographical objects in the field view	TIN	CV_TINCoverage	ISO 19123
	Isolines	CV_SegmentedCurveCoverage	ISO 19123
	AdjPolygons	CV_DiscreteSurfaceCoverage	ISO 19123
	GridOfPoints	CV_DiscreteGridPointCoverage	ISO 19123
	GridOfCells	CV_GridCell	ISO 19123
	IrregularPoints	CV_DiscretePointCoverage	ISO 19123
Network elements	Node	TP_Node	ISO 19107
	Arc	TP_Edge	ISO 19107
	UnidirectionalArc	TP_DirectedEdge	ISO 19107
	BidirectionalArc	TP_DirectedEdge	ISO 19107
Temporal objects	TemporalObject	TM_Object	ISO 19108
	Instant	TM_Instant	ISO 19108
	Interval	TM_Period	ISO 19108

Figure 8 shows the PIM model resulting from performing transformation on the model shown in Fig. 7. The spatial features were transformed into attributes whose types are in accordance with the elements of ISO 191xx standards shown in Table 1. For example, the `City` class, which was modeled with the stereotype `<<Polygon>>`, takes on a geometry attribute, denominated geometry, of the type `GM_Surface`. The same was done with the other classes that have spatial features.

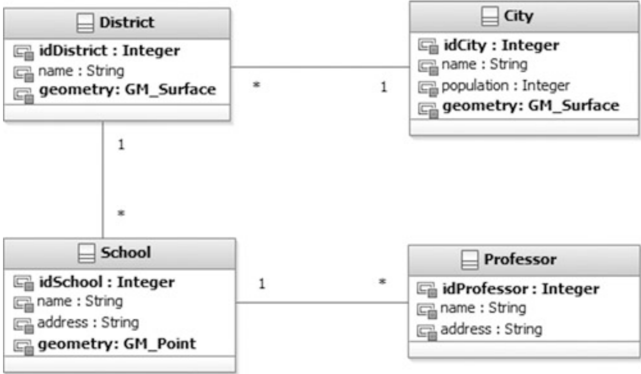


Fig. 8 An example of a model at the PIM level of abstraction

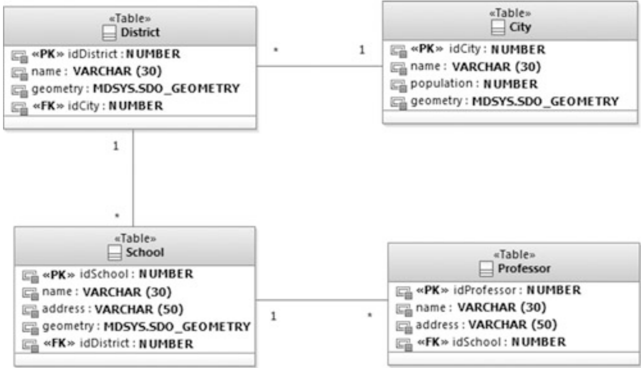


Fig. 9 Example of modeling at the PSM level of abstraction

4.3 PSM Level

The next step is to transform the PIM model into a PSM model, which can be, for example, an object-relational data model extended to manage spatial objects (e.g., Spatial or PostGIS). To illustrate this transformation, Fig. 9 shows an example of the PSM model that corresponds to the platform Oracle Spatial, which was generated from the PIM model shown in Fig. 8. This model already takes into account details of the platform in question, for example, the data types of the platform. Some attributes were also marked with the stereotype `<<PK>>` and `<<FK>>`, which represent the primary and foreign keys, respectively. The purpose of this step is to make the model as close as possible of the chosen platform to automate the generation of the script database.

Listing 1 in the Appendix shows a small part of the transformation code from the CIM model, shown in Fig. 7, to the PIM model presented in Fig. 8, using the ATL models transformation language. The definition of transformations in ATL starts with the transformation module statement as well as the source and

target models. The module is defined using the keyword `module` followed by the module name. The keyword `create` indicates the source and target models [15]. After this step, the transformation rules are defined. Those rules are written using ATL syntax, are saved in files with the extension `.atl`, and can use either a declarative or an imperative style. The code presented in Listing 1 shows one of the transformation rules. This rule is responsible for creating the classes that have geographic information, that in this case are represented by the GeoProfile stereotypes, and for creating the elements that were not contained in the CIM such as, the geometry attribute, whose type need to conform with the ISO standard.

After the transformation of the PIM model, the output model is generated in the XML Metadata Interchange (XMI) format [26], which is a standard format for exchanging UML models among CASE tools.

5 Case Study

This section describes an example of using a customized GeoProfile in the Rational Software Modeler (RSM) CASE tool by IBM®. The study addresses a hypothetical system for managing a sugarcane crop, which has been widely used for biofuel production. This case study was chosen because it could use a large number of elements from the GeoProfile, giving to the reader a good understanding of the profile and how it can be used for domain engineering. Besides, fuel and renewable resources are subjects that are widely discussed nowadays. A brief description of the case study on cultivation of sugarcane for ethanol production is presented below.

The investment in the production of cleaner fuels that can replace, with no economic loss, traditional fuels (e.g., fossil) has been carried out with tax incentives from the Brazilian government. These new fuels, or biofuels, pollute less because the production process tends to be cleaner and have more balanced CO_2 emissions. The main biofuel currently used in Brazil is the ethanol produced from sugarcane. However, its production requires a large amount of natural resources (e.g., areas planted with monoculture) for cultivation and subsequent production of ethanol. The planting, fertilizing, and harvesting (e.g., manual or mechanized), as well as loading, transporting, weighing, unloading and cleaning operations, are crucial for a good industrial performance. Many environmental problems can arise in the production of ethanol, since most crops occupy wide contiguous areas, isolating and/or suppressing forest reserves, as well as the likely deforestation of catchments, siltation of streams, and others.

Based on the above description, one can realize that the problem involves important geographic phenomena that require spatial analysis and therefore need to be stored in a database. As an example, we can mention some natural resources (e.g., soil, topography, vegetation, hydrography) and anthropogenic activities (e.g. production, transport, labor force).

From the description of the problem, a CIM model was initially developed (Fig. 10) using the UML GeoProfile. In that diagram, each class represents an entity of the real world and shows how these can or should be linked. For example, in

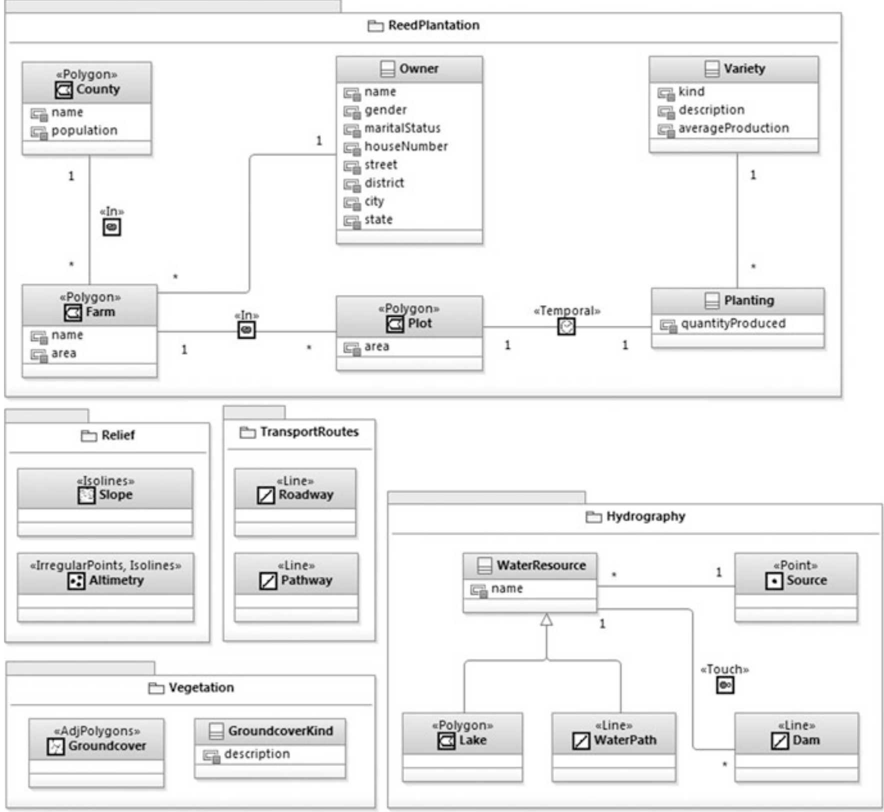


Fig. 10 Conceptual schema (partial) using GeoProfile (the CIM level)

the association element between the classes `Farm` and `Plot`, the specification of a spatial restriction (stereotype `<<In>>`) indicates that the spatial component of each `Plot` should be geometrically within the spatial component of a `Farm`.

Notice that the theme `ReedPlantation` deals with information related to sugarcane farms, such as the location, the division of a farm into plots, varieties that are grown in the farm, and the data on the farm's owner.

Also, in this theme, the temporal association (1:1) between the classes `Plot` and `Planting` indicates that each plot should have only one type of variety, in a given period, but it can record temporal evolutions of the associations of this plot with other types of variety. That is, there cannot be two different sugarcane crops on the same plot in the same period of time.

Data about access roads and some natural phenomena such as topography, vegetation, and hydrology are modeled in specific packages (themes). These phenomena have spatial components, so that information can be retrieved through spatial analysis operations, such as transport routes within a farm, calculation of buffer zones next to water courses, and calculation of slope based on the relief and queries on vegetation types that occur in the area of the farm.

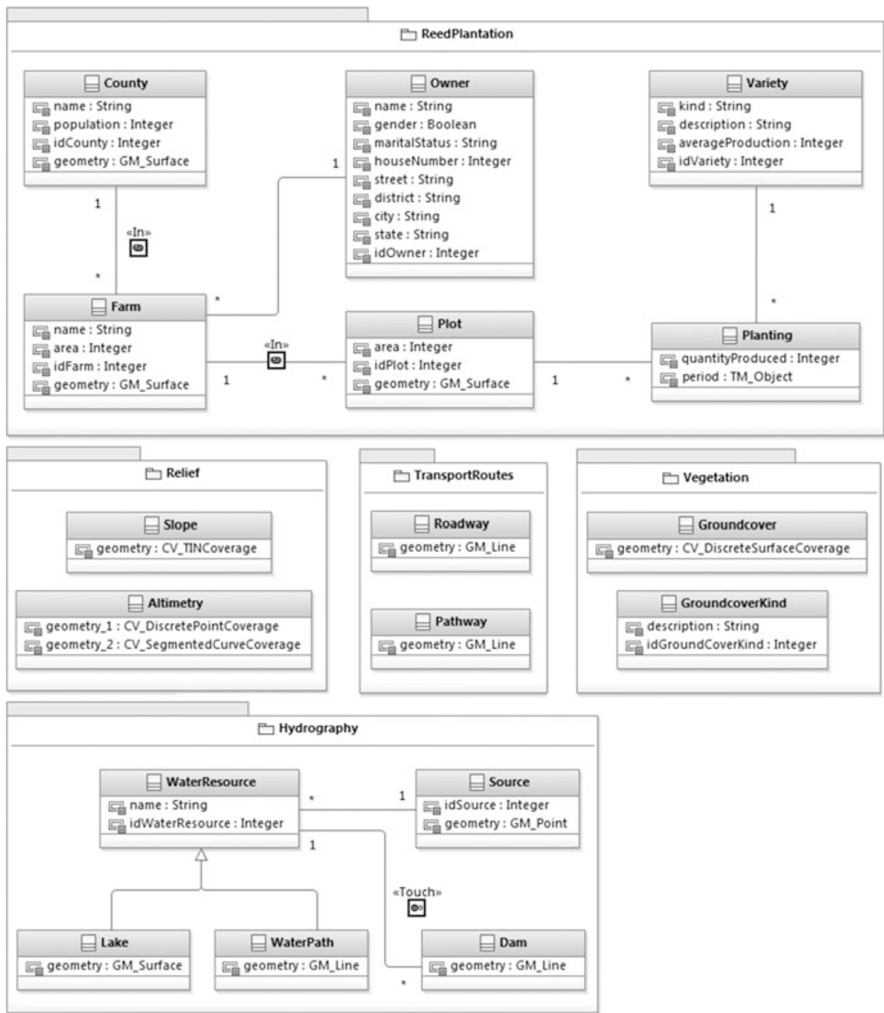


Fig. 11 Schema using the ISO 191xx standards (the PIM level)

The model in Fig. 10, however, is at a high level of abstraction; this is the CIM level in the MDA approach; as seen previously, it is useful for the user and the designer to understand the problem's domain in question. The transformation process of this initial model into a PIM model, the next level of the MDA, follows the same way described in Sect. 4. Figure 11 shows the PIM model resulting from this transformation.

Notice that in this step new specifications were added to the schema. For example, the temporal association (1:1) was transformed into an association (1:*) and one attribute was included in the class `Planting` to store the period in which

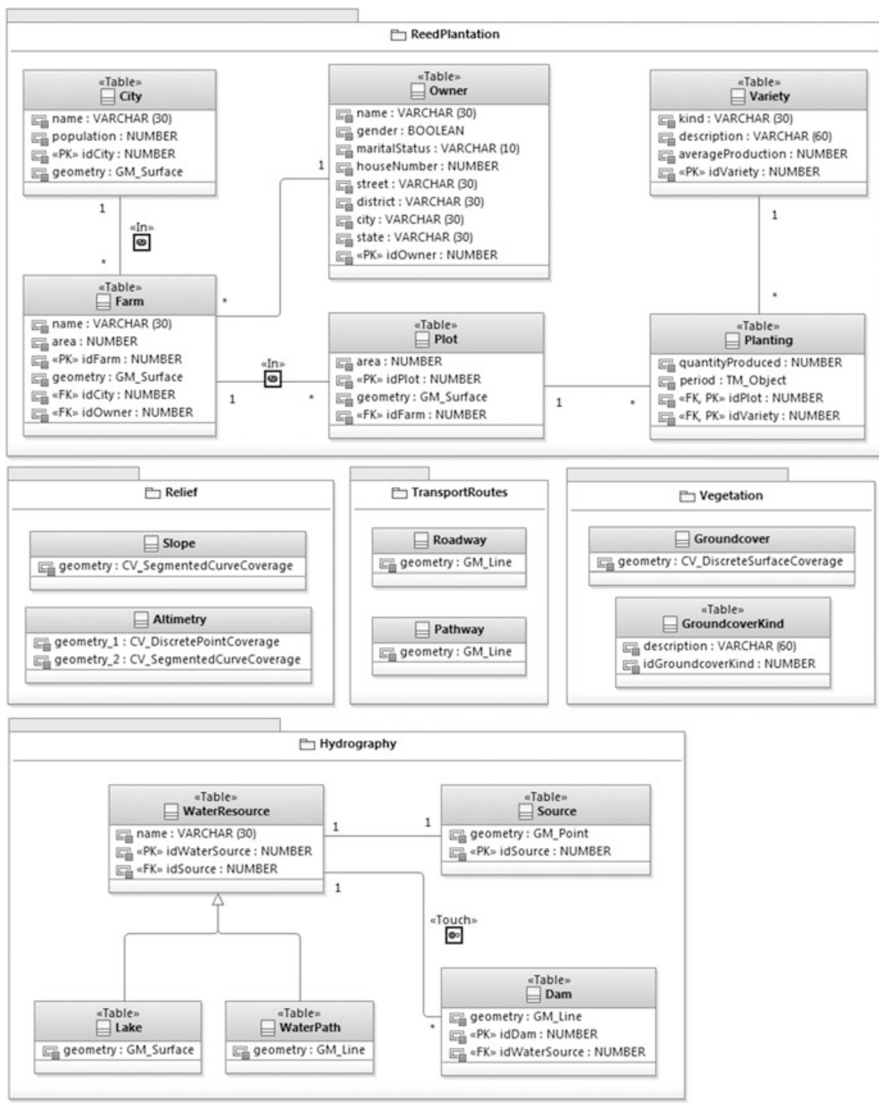


Fig. 12 Customized schema for the object-relational model (the PSM level)

a sugarcane crop was grown in a plot. Attributes related to the geometry and the object identifiers of the classes were also added.

In the next step the PIM is transformed into a PSM model, which is the lowest MDA level. As stated in Sect. 4, a PIM can be transformed into many PSMs, according to the platforms that will be used by the user. However, for this case study, only one example of mapping is shown, namely an object-relational database model. The PSM model resulting from this transformation is shown in Fig. 12.

With the model at the PSM level of abstraction, it is already possible to extract all the information needed to generate the database code. An example of the geographic database script generated from the theme `ReedPlantation` of Fig. 12 using the DBMS Oracle Spatial® is shown in Listing 2 of the appendix.

6 Concluding Remarks

The development of the GeoProfile was mainly motivated by the fact that UML can be used, along with all its available resources, for example, CASE tools, to conceptually model a geographic database. The GeoProfile has in its definition the main requirements for geographic applications and has the features of the main existing conceptual data models for GIS applications.

This chapter showed how GeoProfile meets the international standards for geographic information, using the ISO 191xx standards. The use of standards is essential in the geographic database project. The MDA approach made it possible to show how the GeoProfile is linked to the international standards.

The graphical notation of GeoProfile was customized in the RSM CASE tool, and some examples of conceptual schemes were modeled. The tendency is that the CASE tools, in general, start supporting this mechanism of UML extension, providing a greater number of options for the designer. Finally, a case study was presented, showing how to design a geographic database step by step, using the MDA approach with the aid of a CASE tool that supports a UML2.0. More information about GeoProfile, with examples of how to customize different CASE tools, can be obtained at www.dpi.ufv.br/projetos/geoprofile.

Acknowledgments This project was partially financed by CNPq—National Council for Technical and Scientific Development, MCT—Ministry of Science and Technology and FAPEMIG—Foundation for Research and Development of Minas Gerais.

Appendix

Listing 1 An example of an ATL transformation rule.

```
rule stereotypeClass
{
  from
       input : geoProfile!Class(
          not thisModule.emptyGeometry(input.stereotype))
  to
```

(continued)

(continued)

```
        output : ISO!Class(
           name <- input.name, reference <- input.reference ->
              collect(e | thisModule.getReferences(e)).asSet(),
           attribute <- input.attribute ->
              collect(e | thisModule.getAttributes(e) ).asSet(),
                   attribute <- id,attribute <- geometry
        ),
        id : ISO!Attribute(
           name <- 'id' + input.name,
           type <- thisModule.integerDataType()
        ),
        Geometry : ISO!Attribute(
           name <- input.name + 'Geometry',
           type <- if( thisModule.isPolygon( input.stereotype ))
                   then thisModule.polygonDataType()
                   else thisModule.pointDataType()
                   endif
        )
}
```

Listing 2 Geographic database script generated from the theme ReedPlantation using the DBMS Oracle Spatial® (the PSM level).

```
CREATE TABLE CITY (
  NAME         VARCHAR(30),
  POPULATION   NUMBER,
  IDCITY       NUMBER,
  GEOMETRY     SDO_GEOMETRY,
CONSTRAINT pk_City PRIMARY KEY (IDCITY));
CREATE TABLE OWNER (
  NAME         VARCHAR(30),
  GENDER       VARCHAR(1),
  MARITALSTATUS  VARCHAR(10),
  HOUSENUMBER NUMBER,
  DISTRICT     VARCHAR(30),
  CITY         VARCHAR(30),
  STATE        VARCHAR(30),
  IDOWNER      NUMBER,
CONSTRAINT pk_Owner PRIMARY KEY (IDOWNER));
CREATE TABLE FARM (
  NAME         VARCHAR(30),
  AREA         NUMBER,
  IDFARM       NUMBER,
  GEOMETRY     SDO_GEOMETRY,
  IDCITY       NUMBER,
```

(continued)

(continued)

```
  IDOWNER     NUMBER,
CONSTRAINT pk_Farm PRIMARY KEY (IDFARM),
CONSTRAINT fk_City FOREIGN KEY(IDCITY)
              REFERENCES  CITY(IDCITY),
CONSTRAINT fk_Owner FOREIGN KEY(IDOWNER)
              REFERENCES OWNER(IDOWNER));
CREATE TABLE PLOT (
  AREA        NUMBER,
  IDPLOT      NUMBER,
  GEOMETRY    SDO_GEOMETRY,
  IDFARM      NUMBER,
CONSTRAINT pk_Plot PRIMARY KEY(IDPLOT),
CONSTRAINT fk_Farm FOREIGN KEY(IDFARM) REFERENCES FARM(IDFARM));
CREATE  TABLE  VARIETY (
  KIND        VARCHAR (30),
  DESCRIPTION VARCHAR (30),
  AVERAGEPRODUCTION  NUMBER,
  IDVARIETY   NUMBER,
CONSTRAINT pk_Variety PRIMARY KEY (IDVARIETY));
CREATE TABLE PLANTING (
  QUANTITYPRODUCED   NUMBER,
  GEOMETRY    SDO_GEOMETRY,
  IDPLOT      NUMBER,
  IDVARIETY   NUMBER,
CONSTRAINT pk_Planting PRIMARY KEY (IDPLOT, IDVARIETY),
CONSTRAINT fk_Plot FOREIGN KEY(IDPLOT) REFERENCES PLOT(IDPLOT),
CONSTRAINT fk_Variety FOREIGN KEY (IDVARIETY)
REFERENCES VARIETY (IDVARIETY) );
```

References

1. Bédard, Y., Larrivée, S.: Modeling with pictogrammic languages. In: Shekhar, S., Xiong, H. (eds.) Encyclopedia of GIS, pp. 716–725. Springer, New York (2008)
2. Bédard, Y., Paquette, F.: Extending entity/relationship formalism for spatial information systems. In: Proc. 9th Int. Symp. on Computer-Assisted Cartography, pp. 818–827. Auto-Carto, Baltimore, USA (1989)
3. Belussi, A., Negri, M., Pelagatti, G.: Geouml: a geographic conceptual model defined through specialization of iso tc211 standards. In: 10th EC GI & GIS Workshop, ESDI State of the Art, pp. 1–10. GIS European Commission, Warsaw (2004)
4. Borges, K.A.V., Davis, C.A., Laender, A.H.F.: Omt-g: An object-oriented data model for geographic applications. GeoInformatica **5**(3), 221–260 (2001)
5. Brodeur, J., Badard, T.: Modeling with iso 191xx standards. In: Shekhar, S., Xiong, H. (eds.) Encyclopedia of GIS, pp. 705–716. Springer, New York (2008)
6. Brodeur, J., Bedard, Y., Proulx, M.J.: Modeling geospatial application databases using uml-based repositories aligned with international standards in geomatics. In: Proceedings of the ACM International Symposium on Advances in Geographic Information Systems, pp. 39–46. ACM, Washington, D.C. (2000)

7. Clementini, E., Felice, P.D., Oosterom, P.v.: A small set of formal topological relationships suitable for end-user interaction. In: Proceedings of the Third International Symposium on Advances in Spatial Databases, SSD '93, pp. 277–295. Springer, London (1993)
8. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 6th edn. Addison-Wesley, Boston, MA (2010)
9. Eriksson, H.E., Penker, M., Fado, D.: UML 2 Toolkit. Wiley, New York (2003)
10. Falbo, R.d.A., Guizzardi, G., Duarte, K.C.: An ontological approach to domain engineering. In: SEKE '02: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, pp. 351–358. ACM, New York (2002)
11. Friis-Christensen, A., Tryfona, N., Jensen, C.S.: Requirements and research issues in geographic data modeling. In: Proceedings of the ACM International Symposium on Advances in Geographic Information Systems, pp. 2–8. ACM, Atlanta, Georgia (2001)
12. Goodchild, M.F., Yuan, M., Cova, T.J.: Towards a general theory of geographic representation in gis. Int. J. Geogr. Inform. Sci. **21**(3), 239–260 (2007)
13. IBM: Rational Software Modeler. Accessed Jan 2012. http://www-01.ibm.com/software/awdtools/modeler/(2012)
14. Jensen, C.S.: A consensus glossary of temporal database concepts. ACM SIGMOD **23**, 52–64 (1994)
15. Jouault, F., Kurtev, I.: Transforming models with atl. In: Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science, vol. 3844, pp. 128–138. Springer, Berlin (2006)
16. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing, Boston, MA (2003)
17. Kösters, G., Pagel, B.U., Six, H.W.: Gis-application development with geoooa. Int. J. Geogr. Inform. Sci. **11**(4), 307–335 (1997)
18. Kresse, W., Fadaie, K.: ISO Standards for Geographic Information, 322 pp. Springer, Berlin (2004)
19. Lisboa-Filho, J., Iochpe, C.: A study about data conceptual models for geographic database design. Informática Pública **1**, 67–90 (1999)
20. Lisboa-Filho, J., Iochpe, C.: Modeling with a uml profile. In: Shekhar, S., Xiong, H. (eds.) Encyclopedia of GIS, pp. 691–700. Springer, New York (2008)
21. Lisboa-Filho, J., Sampaio, G.B., Nalon, F.R., Borges, K.A.V.: A uml profile for conceptual modeling in gis domain. In: Proceedings of the International Workshop on Domain Engineering at CAiSE, pp. 18–31. CEUR, Hammamet, Tunisia (2010)
22. Miralles, A., Libourel, T.: Modeling with enriched model driven architecture. In: Shekhar, S., Xiong, H. (eds.) Encyclopedia of GIS, pp. 700–705. Springer, New York (2008)
23. Nalon, F.R., Lisboa-Filho, J., Braga, J.L., de Vasconcelos Borges, K.A., Andrade, M.V.A.: Applying the model driven architecture approach for geographic database design using a uml profile and iso standards. J. Inform. Data Manag. **2**(2), 171–180 (2011)
24. OBEO, A.: Atlas Transfomation Language. Accessed Jan 2012. http://www.eclipse.org/atl/(2012)
25. Object Management Group: MDA Guide, v.1.0.1, OMG Document formal/2003-06-01 edition. OMG, Needham (2003)
26. Object Management Group: Unified Modeling Language: Superstructure, v.2.1.2, OMG Document formal/2007-11-02 edition. Needham (2007)
27. Parent, C., Spaccapietra, S., Zimányi, E.: Modeling and multiple perceptions. In: Shekhar, S., Xiong, H. (eds.) Encyclopedia of GIS, pp. 682–690. Springer, New York (2008)
28. Pinet, F.: Entity-relationship and object-oriented formalisms for modeling spatial environmental data. Environ. Model. Software **33**, 80–91 (2012)
29. Pinet, F., Duboisset, M., Soulignac, V.: Using uml and ocl to maintain the consistency of spatial data in environmental information systems. Environ. Model. Software **22**(8), 1217–1220 (2007)
30. SPARXSYSTEMS: Enterprise Architect. Accessed Jan 2012. http://www.sparxsystems.com/products/ea/.(2012)

31. TC211: ISO 19108: Geographic Information - Temporal Schema. ISO, Geneva (2002)
32. TC211: ISO 19107: Geographic Information - Spatial Schema. ISO, Geneva (2003)
33. TC211: ISO 19123: Geographic Information - Schema for Coverage Geometry and Functions. ISO, Geneva (2005)
34. TC211: ISO: Standards Guide. ISO, Geneva (2009)
35. Worboys, M., Duckham, M.: GIS: A Computing Perspective, 2nd edn. CRC Press, Boca Raton, FL (2004)

Index

A
Abstraction, vii–x, 10, 93, 126, 136, 180, 190, 212, 240, 270, 292, 297, 318, 356, 376
Abstract syntax. *See* Syntax
Agent-oriented modelling, 306, 311
Aggregation, 37, 69, 116, 119, 220, 297, 329, 381
Agile, 183–184, 264, 286–287
Amalgamation, 177
Analogy construction, 116
Artefact, 4, 191, 265, 276, 278, 303. *See also* Artifacts
Artifacts, ix, 30, 62, 84, 114, 134, 216, 240, 318, 351, 381. *See also* Artefact
  application artifacts, 84
  domain artifacts, 84
  product artifacts, 56, 84
Aspect models, 85
Aspect-oriented software development, 85
Aspects
  domain aspects, 85
AToM3, 212, 232, 240
Axiomatization, 326

B
Base models, 85, 178

C
CASE. *See* Computer Aided Software Engineering
Category theory, 269
Clabjects, 138
Classification, 32, 37, 93, 119, 175, 297, 300, 333, 378
Collaboration, 139, 167, 240, 264, 266
Collaborative model editing, 242, 244
Commonality, vi, 4, 84, 119, 264
Common Variability Language, 178
Computation independent model, 386
Computer aided software engineering, 107, 190, 240, 267, 381. *See also* Modeling tools and Software tools
Computer Supported Cooperative Work, 241
Conceptualization, 318
Conceptual model, ix, 136, 302, 330, 351, 378
Concrete syntax. *See* Syntax
Configuration, 4, 30, 116, 134, 193, 288
Consistency, 4, 60, 62, 107, 191, 240, 244, 276, 311
Controller pattern, 61, 75
Cooperative work, 241. *See also* Computer Supported Cooperative Work
Core assets, vi, 84
CSCW. *See* Computer Supported Cooperative Work
CVL. *See* Common Variability Language

D
Denotational semantics, 193, 267
Design patterns, vi, 60, 114, 235
DEVS. *See* Discrete Event System Specification
Domain, v
Discrete Event System Specification, 227
Domain artifacts, 84
Domain aspects, 85
Domain engineering, v, 4, 84, 114, 188, 265, 292, 351, 376
Domain expert, 126, 138, 150, 173, 218, 267
Domain model, 63, 98, 119, 194, 376

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,
DOI 10.1007/978-3-642-36654-3,© Springer-Verlag Berlin Heidelberg 2013

Domain ontology. *See* Ontology
Domain-Specific Language Engineering, v, 190
Domain-specific languages, vii, 23, 114, 160, 188, 212, 254, 269, 292, 319, 355
  external DSL, 126
  formal DSL, 168, 269
  internal DSL, 127
Domain-Specific Modelling, 213, 239
Domain-Specific Modelling Languages, 133, 169, 239, 266, 292, 306, 309–311, 344, 369
Domain-Specific Models, 190, 256
Domain-specific pattern, 60
DSL. *See* Domain-Specific Languages
DSLE. *See* Domain-Specific Language Engineering
DSM. *See* Domain-Specific Models or Domain-Specific Modelling
DSML. *See* Domain-Specific Modeling Languages

E
Eclipse Modeling Framework, 169, 243, 361
EMF. *See* Eclipse Modeling Framework
Evaluation, 128, 134, 169, 317–345
External DSL, 126

F
FBC. *See* Feature-based configuration
Feature, vi, 4, 30, 60
  intrinsic features, 138
Feature-based configuration, 4
Feature diagram 88, 180. *See also* Feature model
Feature extension, 30
Feature intension, 30
Feature location, 30
  dynamic feature location, 47
  static feature location, 39
Feature model, 4, 91. *See also* Feature diagram
Feature-oriented, vi, 29, 84
FM. *See* Feature model
Formal concept analysis, 33
Formal DSL. *See* Domain-Specific Languages
Foundational ontology, 327

G
Generalization, 52, 69, 197, 280, 297, 335
Geographic database, 377
Geographic information system, 377
Graphic designer, 153

H
Hyper-link induced topic search, 36

I
Instantiation, 60, 69, 91, 116, 146, 219, 270, 280, 296, 299, 300, 358
Internal DSL, 127
Interoperability, viii, 23, 165, 188, 241, 273, 350, 376
Intrinsic features, 138
ISO/IEC 24744. *See* Software development methodologies

J
Jargon, 273

L
Language designer, 150, 164
Language engineering, 189, 190, 192, 235, 320. *See also* Domain-Specific Language Engineering
Language specification, 145, 292, 368
Latent semantic indexing, 34
Lean production, 264, 287
Logical models, 325

M
MARTE, 70
Mathematics of modelling, 301
MDA. *See* Model-Driven Architecture
MDD. *See* Model-Driven Development
Message sequence charts, 174
Meta-language, 188, 270
Meta-model, 71, 94, 119, 137, 169, 212, 240, 294, 326, 351, 376
  pivot meta-model, 353
Meta-model extension, 354, 365
Meta-modelling, 135, 137, 212, 292, 294, 329, 353
Meta-object facility, 179, 217, 301, 354
Meta-properties, 329–342, 345
Meta-tool, 161, 191
Method engineering, 134, 256
Model-driven architecture, x, 216, 386
Model-driven development, 83, 351, 376–377
Model-driven engineering, 212, 239. *See also* Domain-Specific Modeling Language or Unified Modeling Language
Model heterogeneity, 350

Model migration, 241, 253
Model quality, 134, 285
Model theory, 268
Model transformation, ix, 78, 212, 228, 351
Model weaving. *See* Weaving
Modeling language. *See* Unified Modeling Language or Domain-Specific Modeling Language
Modeling tools, 137, 171, 243, 350. *See also* Computer-Aided Software Engineering and Software tools
Modular construction, 269
MOF. *See* Meta-object facility
MoSiS, 169
MoTif, 228
MSC. *See* Message sequence charts
Multi-paradigm modeling, 213

N
Notation, x, 21, 60, 84, 134, 150, 160, 189, 213, 268, 302, 307, 311, 335, 363, 386

O
Object constraint language, 61, 71, 193, 219, 385
OCL. *See* Object constraint language
Ontology, 135, 190, 292, 317–345, 352
  domain ontology, 292, 329, 333, 335
  foundational ontology, 327
  reference ontology, 135, 318, 353
Operation-based conflict detection, 251
Operation-based model comparison, 251

P
Pattern representation, 61
Patterns, vi, 100, 116, 217, 272, 307, 352. *See also* Design patterns
  controller pattern, 61, 75
Perspective, 9, 141, 171, 189, 264, 273, 299, 324
Petri net, 225
Pivot meta-model, 353
Platform independent model, 386
Platform specific model, 386
Portability, 164, 387
Powerset, 299
Powertypes, 138, 298
Product artifacts, 56, 84
Productivity, 115, 133, 212, 267, 388
Product lines, v, 4, 22, 30, 84, 178, 192
Profile, vi, 61, 84, 120, 177, 212, 353, 376, 381
Programming language, vii, 50, 87, 114, 137, 189, 212, 267, 299

R
RAMification, 219
Realization, 37, 116, 137, 180
Real-time, vi, 60, 163, 188
Reconciliation, 9, 240, 251
Reference models, 116, 176, 301, 327, 357
Reference ontology. *See* Ontology
Reusability, 5, 107, 114
Reuse, vi, 4, 30, 68, 83, 114, 144, 173, 285, 292, 351, 376
Reuse mechanisms, 116

S
SDL. *See* Specification and Description Language
Semantic domain, 192, 200, 215, 227, 267
Semantic identity, 268, 272
Semantic mapping, 201, 202, 204, 215, 227
Semantics, viii, 4, 95, 117, 134, 145, 164, 188, 192, 194, 200, 204, 215, 271, 298, 325, 359
  denotational semantics, 193, 267
Semiotic, 269, 304, 335
Separation of concerns, 5, 62, 85
Set theory, 228, 296
Simulation, 161, 226, 329, 359
Slicing, 16, 23, 40, 188
SoC. *See* Separation of Concern
Software development methodologies, 283, 294, 306
Software product line engineering, vi, 30, 87, 114, 267
Software product lines. *See* Product lines
Software reuse. *See* Reuse
Software tools, 187, 195, 267, 270. *See also* Computer-Aided Software Engineering and Modeling tools
Specialization, 51, 93, 116, 144, 170, 198, 280, 299, 331, 380, 383
Specification and Description Language, 163
Standardization, 117, 138, 162, 165, 304, 376
Syntax
  abstract syntax, 6, 145, 171, 192, 194, 197, 203, 215, 283, 302, 320, 355, 359, 362, 364

concrete syntax, 6, 134, 169, 194, 197, 204, 215, 291, 330, 361
Syntax mapping, 198, 204
Systems theory, 266

T
Tacit knowledge, 264
T-core, 220
Term frequency, 36
Tool model, 188, 195, 207
Traceability, 49, 62, 352
Train control language, 160
Transparent, 168, 171, 359
Triangle of meaning, 295
Type conformance, 296

U
UML. *See* Unified modeling language
Unified modeling language, vi, 61, 84, 103, 120, 161, 187, 190, 194, 212, 243, 270, 298, 299, 302, 304, 319, 350, 376
UML profile. *See* Profile

V
Value chain, 273
Value chain analysis, 281
Variability, v, 4, 61, 84, 119, 178, 264
Variants, 57, 65, 88, 119, 182, 189, 223
Variation points, 61, 88, 119, 192
Verification, 4, 123, 216, 350
Viewpoint, 9, 268, 273
Visual complexity, 153

W
Weaving, 85, 104, 351, 352