Richard F. Paige (Ed.)

LNCS 5563

Theory and Practice of Model Transformations

Second International Conference, ICMT 2009
Zurich, Switzerland, June 2009
Proceedings

Springer

Richard F. Paige (Ed.)

Theory and Practice of Model Transformations

Second International Conference, ICMT 2009
Zurich, Switzerland, June 29-30, 2009
Proceedings

Springer

Volume Editor

Richard F. Paige
University of York, Department of Computer Science
Heslington, York, YO10 5DD, United Kingdom
E-mail: paige@cs.york.ac.uk

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, D.3, D.4.8, F.3, I.6, K.6, C.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-642-02407-6 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-02407-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12702183 06/3180 5 4 3 2 1 0

Preface

Models have become essential for supporting the development, analysis and evolution of large-scale and complex IT systems. Models allow different views, perspectives and elements of a system to be captured rigorously and precisely, thus allowing automated tools to manipulate and manage the models. In a full-fledged model-driven engineering (MDE) process, the transformations developed and expressed between models are also key. Model transformations allow the definition and implementation of the operations on models, and also provide a chain that enables the automated development of a system from its corresponding models. Model transformations are already an integral part of any model-driven approach, and there are a number of available model transformation languages, tools, and supporting environments; some of these approaches are now approaching maturity. Nevertheless, much work remains: the research community and industry need to better understand the foundations and implications of model transformations, such as the key concepts and operators supporting transformation languages, their semantics, and their structuring mechanisms and properties (e.g., modularity, composability and parametrization). The effect of using model transformations on organizations and development processes – particularly when applied to ultra-large scale systems, or in distributed enterprises – is still not clear. These issues, and others related to the specification, design, implementation, analysis and experimentation with model transformation, are the focus of these proceedings.

The Second International Conference on Model Transformation (ICMT 2009) was held in late June 2009 in Zurich, Switzerland. As was the case with the 2008 edition, the conference was conducted in collaboration with the TOOLS Europe 2009 conference. ICMT built on the success of the inaugural edition in 2008, and the success of tracks on Model Transformation at the ACM Symposium on Applied Computing (SC): MT 2006 in Dijon, France, and MT 2007 in Seoul, Korea. The second ICMT conference brought together researchers and practitioners to share experiences in using model transformations. ICMT 2009 combined a strong practical focus with the theoretical approach required in any discipline that supports engineering practices.

ICMT 2009 received 67 abstract submissions, of which 61 were submitted as full papers. Each paper was reviewed by at least three Program Committee members. There was a full and rigorous discussion process after which the Program Committee recommended 14 full papers for acceptance, giving an acceptance rate of 23%. Additionally, the Program Committee recommended three short papers for acceptance, inclusion in the proceedings, and presentation at the conference. These shorter papers presented work of considerable promise and interest, which led to substantial discussion at the conference. Submissions and the reviewing process were supported by EasyChair, which greatly facilitated these tasks.

The conference was fortunate to have an invited keynote talk by Benjamin Pierce of the University of Pennsylvania. Additionally, an exciting panel on bidirectional transformation was organized by Krzysztof Czarnecki, Nate Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr and James Terwilliger, and was derived from a GRACE meeting held in Japan in December 2008. We are pleased that the panel organizers were also able to prepare an invited paper summarizing the GRACE meeting, which is included in these proceedings.

I thank all members of the Program Committee, and their designated reviewers, for participating and leading a constructive and supportive reviewing process. Their dedication was instrumental in designing a very high quality conference. I especially thank the ICMT 2009 Publicity Chair, Dennis Wagelaar, for his tireless efforts in advertising the conference. I also thank the TOOLS Europe 2009 organizers, particularly Manuel Oriol (TOOLS 2009 Program Chair), local organizers Claudia Günthart and Ilinca Ciupa at ETH Zurich, and Bertrand Meyer (Conference Chair) for their support and help with logistics. The success of ICMT 2009 is greatly due to their efforts. Finally, I thank the ICMT Steering Committee – Antonio Vallecillo, Alfonso Pierantonio, Jeff Gray and Jean Bézivin – for their support and help throughout the process. Without their efforts, ICMT would be but a thought!

June 2009 Richard Paige

Organization

Conference Commitee

Program Chair:	Richard Paige (University of York, UK)
Publicity Chair:	Dennis Wagelaar (V.U. Brussel, Belgium)
Steering Committee:	Alfonso Pierantonio (University of dellAquila, Italy)
	Antonio Vallecillo (University of Málaga, Spain)
	Jeff Gray (University of Alabama at Birmingham, USA)
	Jean Bézivin (INRIA, Nantes, France)

Program Committee

Orlando Avila-Garcia	Open Canarias, Spain
Luciano Baresi	U.P. Milan, Italy
Jordi Cabot	University of Toronto, Canada
Charles Consel	INRIA/LaBRI, France
Davide Di Ruscio	University of Aquila, Italy
Jean-Marie Favre	University of Grenoble, France
Piero Fraternali	Polytechnic Unversity of Milan, Italy
Jesús García-Molina	University of Murcia, Spain
Martin Gogolla	University of Bremen, Germany
Jeff Gray	University of Alabama at Birmingham, USA
Reiko Heckel	University of Leicester, UK
Howard Ho	IBM Almaden, USA
Frédéric Jouault	INRIA, Nantes, France
Gerti Kappel	Technical University of Vienna, Austria
Günter Kniesel	University of Bonn, Germany
Dimitrios Kolovos	University of York, UK
Thomas Kuehne	Victoria University Wellington, New Zealand
Vinay Kulkarni	Tata, India
Ivan Kurtev	University of Twente, The Netherlands
Esperanza Marcos	University of Rey Juan Carlos, Spain
Marc Pantel	University of Toulouse, France
Francesco Parisi-Presicce	University of Rome La Sapienza, Italy

Table of Contents

Invited Paper

Foundations for Bidirectional Programming 1
Benjamin C. Pierce

Full Papers

Model Superimposition in Software Product Lines 4
Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner

Efficient Model Transformations by Combining Pattern Matching Strategies 20
Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró

Managing Dependent Changes in Coupled Evolution 35
Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio

Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages 52
Iván García-Magariño, Jorge J. Gómez-Sanz, and Rubén Fuentes-Fernández

A Collection Operator for Graph Transformation 67
Roy Grønmo, Stein Krogdahl, and Birger Møller-Pedersen

Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions 83
Esther Guerra, Juan de Lara, and Fernando Orejas

Towards Model Round-Trip Engineering: An Abductive Approach 100
Thomas Hettel, Michael Lawley, and Kerry Raymond

Rewriting Queries by Means of Model Transformations from SPARQL to OQL and Vice-Versa 116
Guillaume Hillairet, Frédéric Bertrand, and Jean Yves Lafaye

Solving Constraints in Model Transformations 132
Andreas Petter, Alexander Behring, and Max Mühlhäuser

Automatic Model Generation Strategies for Model Transformation Testing 148
Sagar Sen, Benoit Baudry, and Jean-Marie Mottu

A Simple Game-Theoretic Approach to Checkonly QVT Relations 165
Perdita Stevens

Supporting Model–Driven Development of Object–Relational Database Schemas: A Case Study 181
Juan Manuel Vara, Belén Vela, Verónica Andrea Bollati, and Esperanza Marcos

Typing in Model Management 197
Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière

Supporting Parallel Updates with Bidirectional Model Transformations 213
Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi

Short Papers

Experiments with a High-Level Navigation Language 229
Jesús Sánchez Cuadrado, Frédéric Jouault, Jesús García Molina, and Jean Bézivin

Using Metrics for Assessing the Quality of ASF+SDF Model Transformations 239
Marcel F. van Amstel, Christian F.J. Lange, and Mark G.J. van den Brand

Achieving Rule Interoperability Using Chains of Model Transformations 249
Marcos Didonet Del Fabro, Patrick Albert, Jean Bézivin, and Frédéric Jouault

Panel on Bidirectional Transformations

Bidirectional Transformations: A Cross-Discipline Perspective: GRACE Meeting Notes, State of the Art, and Outlook 260
Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger

Author Index 285

Foundations for Bidirectional Programming

Benjamin C. Pierce

University of Pennsylvania

Most programs get used in just one direction, from input to output. But sometimes, having computed an output, we need to be able to *update* this output and then "calculate backwards" to find a correspondingly updated input. The problem of writing such *bidirectional transformations*—often called *lenses*—arises in applications across a multitude of domains and has been attacked from many perspectives [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16, etc.]. See [17] for a detailed survey.

The Harmony project at the University of Pennsylvania is exploring a *linguistic* approach to bidirectional programming, designing domain-specific languages in which every expression simultaneously describes both parts of a lens. When read from left to right, an expression denotes an ordinary function that maps inputs to outputs. When read from right to left, it denotes an "update translator" that takes an input together with an updated output and produces a new input that reflects the update. These languages share some common elements with modern functional languages—in particular, they come with very expressive type systems. In other respects, they are rather novel and surprising.

We have designed, implemented, and applied bi-directional languages in three quite different domains: a language for bidirectional transformations on trees (such as XML documents), based on a collection of primitive bidirectional tree transformation operations and "bidirectionality-preserving" combining forms [17]; a language for bidirectional views of relational data, using bidirectionalized versions of the operators of relational algebra as primitives [18]; and, most recently, a language for bidirectional string transformations, with primitives based on standard notations for finite-state transduction and a type system based on regular expressions [19,20]. The string case is especially interesting, both in its own right and because it exposes a number of foundational issues common to all bidirectional programming languages in a simple and familiar setting. We are also exploring how lenses and their types can be enriched to embody privacy and integrity policies [21].

This talk surveys some lessons learned from the work so far on Harmony, focusing on foundational issues and attempting to connect them to work ongoing in the model transformation community.

References

1. Meertens, L.: Designing constraint maintainers for user interaction (1998) (manuscript)
2. Kennedy, A.J.: Functional pearl: Pickler combinators. Journal of Functional Programming 14(6), 727–739 (2004)

3. Benton, N.: Embedded interpreters. Journal of Functional Programming 15(4), 503–542 (2005)
4. Ramsey, N.: Embedding an interpreted language using higher-order functions and types. In: ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators (IVME), San Diego, CA, pp. 6–14 (2003)
5. Hu, Z., Mu, S.C., Takeichi, M.: A programmable editor for developing structured documents based on bi-directional transformations. In: Partial Evaluation and Program Manipulation (PEPM), pp. 178–189 (2004); Extended version to appear in Higher Order and Symbolic Computation (2008)
6. Brabrand, C., Møller, A., Schwartzbach, M.I.: Dual syntax for XML languages. In: Information Systems (2007) (to appear); Extended abstract in Database Programming Languages (DBPL) (2005)
7. Kawanaka, S., Hosoya, H.: bixid: a bidirectional transformation language for XML. In: ACM SIGPLAN International Conference on Functional Programming (ICFP), Portland, Oregon, pp. 201–214 (2006)
8. Fisher, K., Gruber, R.: PADS: a domain-specific language for processing ad hoc data. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, IL, pp. 295–304 (2005)
9. Alimarine, A., Smetsers, S., van Weelden, A., van Eekelen, M., Plasmeijer, R.: There and back again: Arrows for invertible programming. In: ACM SIGPLAN Workshop on Haskell, pp. 86–97 (2005)
10. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
11. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Transactions on Database Systems 6(4), 557–575 (1981)
12. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM Transactions on Database Systems (TODS) 13(4), 486–524 (1988)
13. sGreenberg, M., Krishnamurthi, S.: Declarative Composable Views, Undergraduate Honors Thesis. Department of Computer Science, Brown University (2007)
14. Voigtländer, J.: Bidirectionalization for free! In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Savannah, Georgia, pp. 165–176 (January 2009)
15. Lutterkort, D.: Augeas–A configuration API. In: Linux Symposium, Ottawa, ON, pp. 47–56 (2008)
16. Hu, Z., Mu, S.C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. Higher-Order and Symbolic Computation 21(1-2) (June 2008); Short version in PEPM 2004
17. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. ACM Transactions on Programming Languages and Systems 29(3), 17 (May 2007); Extended abstract in Principles of Programming Languages, POPL (2005)
18. Bohannon, A., Vaughan, J.A., Pierce, B.C.: Relational lenses: A language for updateable views. In: Principles of Database Systems, PODS (2006); Extended version available as University of Pennsylvania technical report MS-CIS-05-27

19. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California (January 2008)
20. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. In: ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, British Columbia (September 2008)
21. Foster, J.N., Pierce, B.C., Zdancewic, S.: Updatable security views. In: Computer Security Foundations Symposium (2009)

Model Superimposition in Software Product Lines

Sven Apel[1], Florian Janda[1], Salvador Trujillo[2], and Christian Kästner[3]

[1] Department of Informatics and Mathematics, University of Passau, Germany
{apel,janda04}@uni-passau.de
[2] IKERLAN Research Centre, Spain
STrujillo@ikerlan.es
[3] School of Computer Science, University of Magdeburg, Germany
ckaestne@ovgu.de

Abstract. In software product line engineering, feature composition generates software tailored to specific requirements from a common set of artifacts. Superimposition is a technique to merge code pieces belonging to different features. The advent of model-driven development raises the question of how to support the variability of software product lines in modeling techniques. We propose to use superimposition as a model composition technique in order to support variability. We analyze the feasibility of superimposition for model composition, offer corresponding tool support, and discuss our experiences with three case studies (including an industrial case study).

1 Introduction

Modeling is essential to deal with the complexity of software systems during their development and maintenance. Models allow engineers to precisely capture relevant aspects of a system from a given perspective and at an appropriate level of abstraction. Initially, modeling in software development aimed at the description of single software systems. Typically, for each software system there is a set of models that describe its static structure, dynamic behavior, interaction with the user, and so on. With 'model' we refer henceforth to a concrete software artifact written in a modeling language that describes a certain facet of a software system.

Recently, researchers and practitioner have realized the necessity for modeling variability of software systems [1, 2, 3, 4]. Especially, software product line engineering poses major challenges on contemporary modeling techniques [5]. A *software product line* is a set of software intensive systems that are tailored to a specific domain or market segment and that share a common set of features [1,6,7]. A *feature* is an end-user visible behavior of a software systems, and features are used to distinguish different software systems, a.k.a. *variants*, of a software product line [1]. For example, in telecommunication software, automatic callback and an answering machine are two features that are not necessarily present in all possible telecommunication systems.

There are two facets of modeling in software product lines. First, there are approaches for describing the variability of a product line, i.e., they specify which feature combinations produce valid variants [1]. Second, all variants of a product line may have models that describe them. Since the variants have usually significant overlaps in their functionality, their models have significant overlaps, too, and it is desirable to factor out these overlaps. So, modeling languages have to take this into account.

We aim at the latter facet of modeling. The basic idea is to decompose a (structural, behavioral, etc.) model according to the features of the product line. This allows us to generate models for individual variants by composing model fragments, instead of maintaining individual models for every variant. A *model fragment* is a part of a model that covers only a feature of a system. Apart from that, a model fragment has to be syntactically complete according to its metamodel. Two things are needed for that: (1) a means to express model fragments and (2) a mechanism to compose model fragments in different combinations yielding different variants. For example, in our telecommunication system, we have a base system and two features. Each of them contains a class diagram (state diagram, activity diagram, and so on) that captures precisely the part of the complete model that is 'added' by the feature. When generating a specific telecommunication system variant, the engineer selects the desired features and then the corresponding models are composed by a generator.

The decomposition of models into fragments (features) solves two problems in modeling, *complexity* and *variability* [2,3,8,9]: (1) engineers tame complexity by modeling only parts of a possibly large system, and (2) engineers do not provide models for each distinct variants of a software system but they provide model fragments for the system's features, from which the models of the variants are generated. Solutions to both problems are essential in order to increase the productivity of modeling and to let the vision of model-driven development come true.

Many approaches to feature composition rely on the technique of *superimposition* [10, 11, 12, 13, 14]. Superimposition is a relatively simple composition technique that, nevertheless, has been used successfully for the composition of code, written in a wide variety of languages, e.g., Java, C#, C++, C, Haskell, Scheme, JavaCC, and Bali [10,11,14]. So, naturally, the question arises whether superimposition is expressive and powerful enough for the composition of models, especially, in the face of the diverse kinds of models used today.

For the purpose of a compact discussion, we concentrate on three kinds of models supported in the unified modeling language (UML), namely, class diagrams, state diagrams, and sequence diagrams. They differ significantly in their syntax and semantics and thus cover a sufficiently broad spectrum of model elements for our discussion. On the basis of an analysis of the feasibility and expressiveness of model composition with superimposition, we have extended the feature composition tool FEATUREHOUSE for UML model composition. We used FEATUREHOUSE in three case studies on model composition: two academic product lines and a product line of our industrial partner, from which we will report our experiences. The studies demonstrate that, even though superimposition is a simple, syntax-driven composition technique, it is expressive enough for the scenarios we looked at. Furthermore, our analysis and studies reveal some interesting issues w.r.t. the trade-off between expressiveness and simplicity, which we will discuss.

In summary, we make the following contributions: (1) an analysis of superimposition as composition technique for UML models, (2) a tool for UML model composition on the basis of superimposition, and (3) three case studies (including an industrial study) on model superimposition and a discussion of our experiences.

2 Software Product Lines and Feature Composition

A *software product line* is a set of software intensive systems that are tailored to a specific domain or market segment and that share a common set of features [1, 6, 7]. A *feature* is an end-user visible behavior of a software system. Features are used to distinguish different *variants* of a software product line. *Feature composition* is the process of assembling and composing software artifacts (e.g., models) belonging to different features based on a user's feature selection [10, 14].

One popular approach to feature composition is superimposition. *Superimposition* is the process of composing software artifacts by merging their corresponding substructures on the basis of nominal and structural similarity. Superimposition has been applied successfully to the composition of software artifacts in different application scenarios and languages [10, 14, 15, 16].

In recent work, it has been shown that superimposition as feature composition technique is applicable to a wide variety of software artifacts, mostly source code artifacts written in different languages, e.g., Java, C#, C++, C, Haskell, Scheme, JavaCC, and Bali [10, 11, 14]. Before we explore the feasibility and expressiveness of superimposition for model composition, we explain how superimposition is commonly used for composing source code artifacts.

On the left side in Figure 1, we show a simple Java class implementing a rudimentary phone system that allows clients to make outgoing calls. On the right side, we show the abstract structure of the class in the form of a *feature structure tree* (*FST*) [13, 14]. An FST represents the essential modular structure of a software artifact and abstracts from language-specific details. In our example, the FST contains nodes representing the classes Phone and OutCall as well as their members.[1]

Now suppose we want to add a feature that allows users to receive incoming calls. In Figure 2, we show the Java code and a corresponding FST that implement incoming calls. In order to create a phone system that contains both features, outgoing and incoming calls, we superimpose their FSTs. In Figure 3, we show the result of the superimposition of the FSTs of OUTCALLS and INCALLS. The two FSTs are superimposed by merging their nodes, matched by theirs names, types, and relative positions, starting from the root and descending recursively. Their superimposition, denoted by '•', results in a merged package phone containing all three classes Phone, OutCall, and InCall. The class Phone is interesting as it contains the union of the members of its instances in the features OUTCALLS and INCALLS.

Assuming further features like CALLBACK or ANSWERING, we have an SPL from which several telecommunication system variants can be generated, e.g., INCALL • ANSWERING, OUTCALL • CALLBACK, or OUTCALL • CALLBACK • INCALL • ANSWERING. Since not all combinations form meaningful variants, additional constraints define which combinations are valid [17, 18].

[1] Note that typing information is attached to each node and the actual content of the methods is hidden in the leaves of the FST.

```
package phone;
class Phone {
  String number; String owner; int state;
  void outCall(String num) { ... }
}
class OutCall {
  double time; double date;
  void makeCall(String num) { ... }
}
```

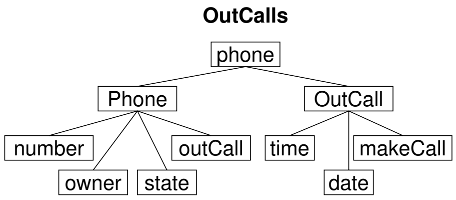


Fig. 1. Java code and FST of the feature OUTCALLS

```
package phone;
class Phone {
  int voiceMsg;
  void inCall() { ... };
}
class InCall {
  double time; double date;
  void acceptCall() { ... }
}
```

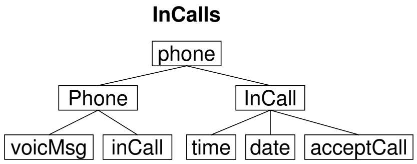


Fig. 2. Java code and FST of the feature INCALLS

```
package phone;
class Phone {
  String number; String owner; int state;
  void outCall(String num) { ... }
  int voiceMsg;
  void inCall() { ... };
}
class OutCall {
  double time; double date;
  void makeCall(String num) { ... }
}
class InCall {
  double time; double date;
  void acceptCall() { ... }
}
```

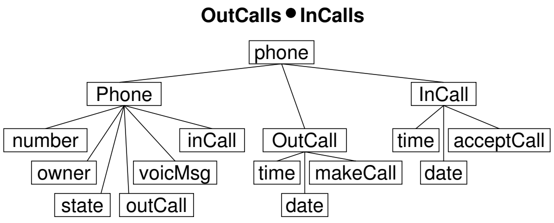


Fig. 3. Java code and FST of the superimposition of OUTCALLS and INCALLS

While superimposition has been used successfully in software product lines [10, 14, 19] and component systems [16], it is not a general technique for all kinds of composition scenarios. A limitation is that the structures of two software artifacts being composed have to obey certain similarities that guide the composition. Accidental name clashes cannot be detected and possibly necessary renamings cannot be performed automatically since the semantics of the underlying artifacts is not considered. We discuss the implications of this limitation for our case studies in Section 6.

As said, it has been shown that superimposition is applicable to a wide variety of software artifacts written in different languages [10, 14]. However, superimposition imposes several constraints on the target language: (1) the substructure of a feature must be a hierarchy of modules, (2) every module (submodule, ...) of a feature must have a name, and (3) the name of a module must be unique in the scope of its enclosing module (that is, a module must not have two submodules with identical names). Typically, these constraints are satisfied by most programming languages, but also other (non-code) languages like XML or grammars align well with them [10, 20, 14]. In the next section, we analyze whether and how superimposition is capable of being used for model composition, especially for the composition of UML models.

3 Analysis of Superimposition as Model Composition Technique

In this section, we analyze whether UML class diagrams, state diagrams, and sequence diagrams, can be decomposed into features and composed again to create complete models corresponding to the variants of an SPL. We do not address layouting issues but only structural composition. In Table 1, we give an overview of our findings, which are conveyed in the remaining section. The table shows for each diagram type (Column 1) the elements that can be composed (Column 2), how elements are identified – if at all (Column 3), possible variants of an element (Column 4), and the composition method, i.e., how two elements are composed to form a new element (Column 5). Note that the difference between superimposition and replacement is that with superimposition the corresponding substructures of two elements are superimposed recursively. With replacement one element substitutes the other completely. Concatenation is used only in sequence diagrams, which is explained below.

Class Diagrams. An UML class diagram consists of a set of packages, entities, and relationships between these entities. Entities are displayed by boxes and may denote plain classes, implementation classes, association classes, interfaces, types, and so on. A package may contain several further packages and entities. Each package and entity has a unique name (possibly consisting of an instance and a class name) in the scope of its enclosing package as well as a type that corresponds to its syntactical category or stereotype (e.g., «class» or «interface»). Relationships are displayed by different kinds of arrows and denote inheritance, aggregation, composition, and so on. Relationships do not necessarily have unique names but may have annotations such as cardinalities or message names.

Table 1. An overview of UML model elements and their composition

diagram	**element**	**name**	**variants (excerpt)**	**composition**
class	package	identifier	—	superimposition
	entity	instance, class	class, interface, type	superimposition
	attribute	type, variable	valued attributes	replacement
	operation	signature	—	replacement
	relationship	optional identifier	generalization, association	replacement
state	state	identifier	—	superimposition
	transition	optional identifier	—	replacement
	start/stop state	—	—	replacement
sequence	role	instance, class	—	superimposition
	lifeline	—	—	concatenation
	interaction	—	—	—

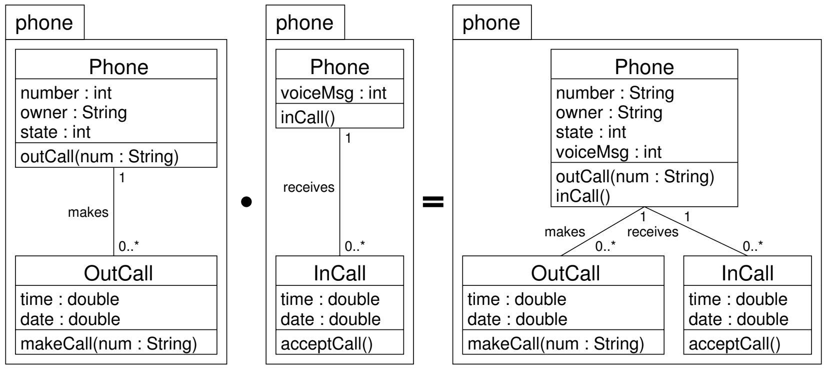


Fig. 4. Superimposition of the class diagrams of OUTCALLS and INCALLS

We illustrate the superimposition of class diagrams by the example of our phone system. In Figure 4, we show from right to left the class diagrams of OUTCALL, INCALL, and their superimposition OUTCALL • INCALL.

During superimposition, entities are matched by name and type. That is, the class Phone in package phone is merged with the other class Phone in package phone stemming from another feature. Entities of different types or from different packages are not composed. When composing two entities, the union is taken from their members, much like in our Java example in Figure 3. If there are two members with the same name (and type), one replaces the other, whose order is inferred from the feature composition order.

Relationships cannot be merged in every case by name since they do not always have names. Implicitly, we assume that each unnamed relationship has a unique name (possibly automatically assigned by a tool) – thereby satisfying the second and third constraint of Section 2. Hence, in our example, both of the two associations from the features OUTCALLS and INCALLS appear in the superimposed diagram. If two relationships have identical names, one replaces the other, much like with members. Other types of entities and relations are treated similarly to the ones explained above.

In summary, the superimposition of class diagrams is straightforward as it resembles the superimposition of FSTs of object-oriented code. Of course, more complex, possibly semantics-driven, composition rules are possible, e.g., for the composition of cardinalities, but for our use cases, a simple replacement of the involved relationships is simpler and sufficient, as we will discuss in Sections 5 and 6.

State Diagrams. An UML state diagram consists basically of a set of states and a set of state transitions. States have unique names and may have inner states. So, they can be composed by name. As with packages, inner states must have unique names in the scope of their enclosing state in order to be superimposed recursively. A state transition may have annotations and a unique name. In case there is no unique name, we assign one, as with relationships in class diagrams. Other elements of state diagrams are handled similarly to states and transitions; if they have names (e.g., signals), they are treated like states; if they do not have unique names (e.g., choices), they are treated like unnamed transitions.

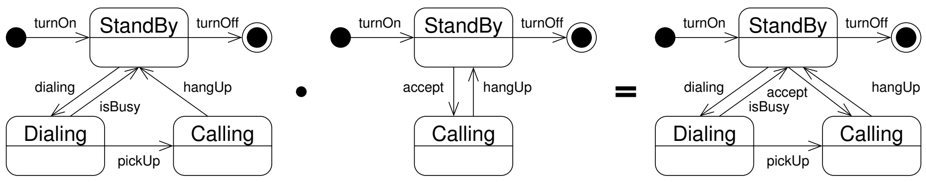


Fig. 5. Superimposition of the state diagrams of OUTCALLS and INCALLS

The start and stop states of a state diagram deserve a special attention. Each state diagram must have exactly one start and possibly one stop state. For composition that means that the start and stop states of one feature replace the start and stop states of the other feature.

In Figure 5, we illustrate the composition of two state diagrams that describe the behavior of our phone system. The state diagrams of OUTCALLS (left) and INCALLS (middle) are superimposed (shown on the right). Essentially, the feature INCALLS adds a new transition from the state StandBy to the state Calling. In our case studies, we found more complex examples in which also compound states and multiple start and stop states are involved.

In summary, the superimposition of state diagrams is similar to class diagrams, except the treatment of start and stop states. States correspond roughly to classes and transitions correspond roughly to relationships. All other elements such as signals and choices can be assigned to one of these two categories. As with class diagrams, more complex composition rules are possible, e.g., for composing choices, but we found our rules are sufficient, as we will discuss in Sections 5 and 6.

Sequence Diagrams. An UML sequence diagram consists of a set of roles that represent time lines of objects and different kinds of interactions that represent the timely interplay between objects. Roles have unique names that consist of object and/or class name. Hence, they can be composed by name, much like classes. Furthermore, each role has a lifeline – displayed from top to bottom – that is the source and destination of interactions with other roles, which are denoted with arrows. Along the lifelines, the time proceeds from top to bottom. Interactions have usually annotations (e.g., operation names and guards) but do not have unique names. Although there may be names, these do not need necessarily to be unique. For example, along its lifeline, a role may interact with another role via multiple interactions that have identical names (e.g., by invoking an operation multiple times), but that occur at different points in time. Hence, we assign unique names (our tool does), much like we do with unnamed relationships in class diagrams.

The composition of two roles with the same name is done by adding the interactions of the second role to the ones of the first role. In Figure 6, we show the composition of the sequence diagrams of the features OUTCALLS (left) and INCALLS (middle). Phone is the only role that is defined in both features. In the superimposed lifeline (right), the Phone's lifeline of feature INCALLS is added below the corresponding lifeline of feature OUTCALLS, which is de facto a concatenation.

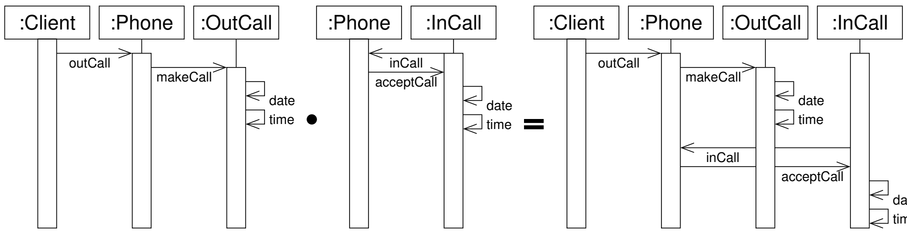


Fig. 6. Superimposition of the sequence diagrams of OUTCALLS and INCALLS

While this approach works for our example, this kind of composition of lifelines may be too restricted. What would we have to do if we want to add the interactions of INCALLS before or somewhere in the middle of the ones of OUTCALLS? While the former case would be possible by reversing the composition order of OUTCALLS and INCALLS, the second case cannot be implemented using superimposition. This problem has been observed in the context of other programming languages and several solutions have been proposed, e.g., splitting features or injecting hooks [14] (see also Sec. 6).

In summary, sequence diagrams can be composed like class diagrams, where roles correspond to classes and interactions correspond to relationships, except that the order of interactions matters in the composition of lifelines. As with class and state diagrams, more complex composition rules would be possible, e.g., for composing guards, but we found our rules are sufficient (see Sec. 5 and 6).

4 Tool Support

Based on our analysis of the feasibility of superimposition for the composition of UML models, we have extended our tool FEATUREHOUSE[2] to enable it to compose UML diagrams. FEATUREHOUSE is a software composition tool chain that relies on feature structure trees and superimposition [14]. In Figure 7, we show the architecture of FEATUREHOUSE. The central tool is FSTCOMPOSER which language-independently superimposes FSTs stored in a proprietary data structure. At certain points during superimposition special composition rules are applied, such as replacement and concatenation, which we have implemented as specified in Table 1.

Furthermore, FSTGENERATOR generates for each language (1) a parser that translates artifacts written in that language to FSTs and (2) a pretty printer that creates an artifact out of an (composed) FST. FSTGENERATOR receives as input an annotated FEATUREBNF grammar (an extended BNF format with annotations that control superimposition) of the language whose artifacts are going to be composed. The details of this generation step are out of scope of this paper and explained elsewhere [14]. We have implemented the rules of superimposing UML diagrams as explained in Section 3.

Technically, we use the XML metadata interchange 1.2 format (XMI) [21] as an representation of UML diagrams and ArgoUML[3] for creating, displaying, and editing UML

[2] http://www.fosd.de/fh

[3] http://argouml.tigris.org/

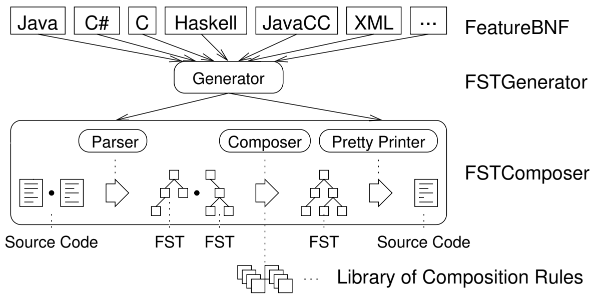


Fig. 7. Architecture of FEATUREHOUSE

diagrams stored in XMI files. Points in favor of XMI are that XMI files are machine-readable by FEATUREHOUSE and can be transformed easily into FSTs. In a first step, we have developed a parser manually (without a generation step) that translates class, state, and sequence diagrams stored in XMI files to FSTs and a pretty printer that translates the (composed) FSTs back to XMI files. In a second step, we have extended FSTGENERATOR by the ability to generate a parser and pretty printer automatically on the basis of an annotated grammar, which was in our case an XML schema document. Interestingly, the annotated grammar plays the role of the metamodel of the models we consider. This raises the question if other more common metamodels such as Ecore could be used instead, which we address in further work.

A typical development cycle consists of four steps: (1) ArgoUML is used to design model fragments belonging to different features; (2) the visual model representations are exported to XMI using ArgoUML's standard export facilities; (3) FEATUREHOUSE is used to compose multiple model fragments based on a user's feature selection; (4) ArgoUML is used to view the composed result. We have used FEATUREHOUSE (and ArgoUML) in three case studies, which we will explain next. For simplicity, we assume that the feature selections passed to FEATUREHOUSE are valid. Existing tools like GUIDSL [18] can be easily used with FEATUREHOUSE to ensure the validity of a feature selection.

5 Case Studies

We have explored the feasibility and practicality of FEATUREHOUSE for model composition by means of three case studies. Specifically, we wanted to know whether such a simple approach like superimposition is expressive enough to compose models in practice. All models (XMI and PNG files) of all case studies can be downloaded from the Web.[4] In order to protect the intellectual property of our industrial partner, we have made the models of the gas boiler system (third case study) anonymous to some extent.

Audio Control System. As a first, simple case study, we have designed an audio control system (ACS). ACS is a small product line consisting only of three features: a basic

[4] http://www.fosd.de/mc/ICMT2009.zip

Table 2. An overview of UML model elements and their composition

class diagrams:

	class		member			relationship		
	all	**extended**	**all**	**added**	**replaced**	**all**	**added**	**replaced**
ACS	12	4	30	7	0	8	3	1
CMS	7	3	28	9	6	4	2	0
GBS	22	8	24	31	61	8	12	4

state diagrams:

	state					transition		
	all	**extended**	**compound**	**start**	**stop**	**all**	**added**	**replaced**
ACS	11	2	2	3	1	25	8	0
CMS	16	7	2	3	3	26	18	2

sequence diagrams:

	role/lifeline		interaction		
	all	**extended**	**all**	**added**	**replaced**
ACS	8	5	3	14	0
CMS	5	6	11	9	0

all: number of all elements in the product line; **extended**: number of all elements that have been extended, possibly multiple times; **added**: number of all elements that have been added to a given program; **replaced**: number of all elements that have been replaced by other elements; **compound/start/stop**: number of compound/start/stop states;

amplifier, a remote control, and a compact disk player. We designed the structure and behavior of ACS with class, state, and sequence diagrams. Overall, we have developed three class diagrams, two state diagrams, and three sequence diagrams, i.e., not for every feature there is a distinct state diagram. From the ACS product line four different variants can be generated which results in different class, state, and sequence diagrams that have been composed by the model fragments of the selected features. In Table 2, we show some numbers about the models of this case study.

The decomposition of ACS into features and their subsequent composition in different combinations was straightforward. This may be due to the simplicity of the case study and/or that we designed it with features in mind. The dominant activity of features was to add new elements and to extend existing elements with subelements, such as to extend classes with members or lifelines with interactions. This case study provides a good canonical example for the feasibility of model composition by superimposition.

Conference Management System. As a second case study, we have decomposed the conference management system (CMS) of [22] into features. The initial version of CMS contained only class diagrams, decomposed into four features: user management, submission system, submission verification procedure, and review process. Based on the informal description of the authors of CMS [22], we complemented the class diagrams with state and sequence diagrams. Although more are possible, we have composed four concrete variants of CMS. Overall, the decomposed version of CMS consists of four class diagrams, four state diagrams, and four sequence diagrams. Like ACS, the decomposition into features and the composition of features in order to generate variants was mostly straightforward. However, CMS is more complex than ACS and was designed by a third party and not designed as an SPL – so it is a more unbiased example. An interesting property of CMS is the use and superimposition of compound states, as can be seen in Table 2. These are superb use cases for superimposition.

Gas Boiler System. As a third case study, we have composed different variants of a gas boiler system (GBS). GBS was developed by a customer of our industrial partner, the IKERLAN Technology Research Centre,[5] Mondragon, Spain, for serving as embedded control software of gas boilers. The developers at IKERLAN have refactored the initial versions of GBS into a software product line. Furthermore, they have used modeling techniques, in particular, UML class diagrams, to describe the structure of the overall system. In contrast to our approach, they have developed a complete class diagram in which model elements of all features are merged, and they use annotations to maintain the relationship between model elements and features.

Based on their annotations, we have decomposed the complete class diagram into 29 class diagram fragments that correspond to the 29 features of GBS. Despite the complexity of this case study, the (de)composition was relatively simple because information about features and variants was available from our partner. Mostly, features add new classes and extend existing classes by new members and relationships. One interesting issue was how to model the variability inside a method of a class. The problem is that class diagrams do not expose details about method bodies and the developers did use notes or other kinds of annotations for that. In their initial approach, they simply annotated a method to be associated to two or more features. This is not possible with superimposition. Instead, we have modeled this kind of *intra-method variability* by introducing a corresponding class and member to every feature that affects this member. For example, if feature A and B both affect the implementation of a method m contained in class C, then the class diagrams of both features define a class C with method m.

5.1 Summary of Experiences

We summarize our experiences in the following list:

- Mostly, the decomposition of models into features and the composition of features for the generation of model variants was straightforward. The predominant "activities" of features were to add new elements such as classes, states, or roles, and to extend existing elements, e.g., by adding new attributes, operations, relationships, transitions, or interactions. Usually, the addition of new elements and the extension of existing elements occurred together in order to connect the newly introduced elements in some way to the existing elements. Extensions at a finer grain, such as changing or extending a guard or an annotation, did not occur, even not in the third-party case studies.
- The fact that two models need to obey certain similarities in their structure in order to be superimposed caused no problems in our case studies. However, for modeling product lines from scratch, as in the case of the ACS study, it is necessary to plan carefully the structure of the base models and their subsequent extensions applied

[5] http://www.ikerlan.es

by features. A decomposition of existing models, as in the GBS case study, naturally leads to models that can be superimposed again in different combinations.
- The problem that roles in sequence diagrams can only be extended by adding new interactions at the end or in front of their lifelines did not occur in our case studies. Usually, the features that extend lifelines add sequences of interactions that are semantically distinct from the existing interactions. Of course, in other cases this problem may be more daunting and has to be explored then.
- We found some convincing use cases of nested model elements, e.g., compound states in CMS, that take advantage of the recursive superimposition process.
- Intra-method variability, i.e., the situation in which multiple features affect the implementation of a single method, occurs in several situations in our industrial case study. The reason is that the developer already thought about the implementation of the model. We handle this case by introducing a corresponding model element into each feature that is involved.
- Cases for the replacement of existing elements did not occur frequently. We found only use cases for relationships in which two identical relationships have been composed. The reason was readability, i.e., both class diagram fragments were easier to understand with the relationship in question.

6 Discussion: Simplicity vs. Expressiveness

Superimposition is a comparatively simple composition technique. Not least this is rooted in its aim at generality. Based on our experience with superimposition of source code artifacts, we have explored if it is applicable to model composition. Since models are mostly of a hierarchical structure, superimposition is a good match, as it is indicated by our examples and case studies. Due to our focus on software product lines and feature-based (de)composition, the input models naturally have similar structures so that they could be easily superimposed. In other scenarios, such as multi-team development, additional refactorings may be necessary.

However, applicability is not the only criterion. Is superimposition expressive enough to compose models in practice? Many model transformation and composition techniques aim at more powerful, fine-grained, and semantics-based composition models [2,3,23,24,25,26,27]. With these techniques, even elements such as guards or cardinalities can be composed, elements can be renamed and changed in manifold ways, and semantic constraints check the correctness of a transformation/composition. Superimposition is syntax-driven without semantic checks and there are not many ways to change an element (replacement and concatenation). However, erroneous compositions have been rejected by analyzing a feature model that defines the valid feature combinations of an SPL [18]. Additionally, experts from our industrial partner checked the results of our compositions for correctness. But it is certainly desirable to automate this process using constraint-based techniques, e.g., [25]. However, at least for our case studies, superimposition was expressive enough to satisfy the composition demands of the considered applications scenarios in SPL development. Especially, the industrial setting of the GBS study indicates that, with superimposition, we can go a long way.

Important to note is that superimposition is no technique for the integration of arbitrary models. It is useful for decomposing models into features, where each feature-related model fragment has a certain structural similarity with other model fragments in order to be superimposed. Automated renamings, rebindings, and refactorings are not supported. Thus, superimposition is not the "silver bullet", but useful at least in the context of feature composition and software product lines. Also the success of superimposition for the composition of code is a motivation for our work. We believe that a general model for feature composition (for source code, models, documentation, makefiles, test cases, etc.) helps the programmer to tame complexity and to gain insight into the software since all software artifacts are treated uniformly during composition.

Our work suggests that there is a trade-off between expressiveness (fine-grained, semantics-based composition) and simplicity (superimposition). Of course, we cannot judge for one or the other, nor infer an ideal mixture of both. This trade-off has been discussed for years in the programming languages community and led to several interesting approaches (just think of the difference in complexity of Scheme and Haskell); we believe that our work can help to initiate a discussion about this issue for modeling languages and model composition mechanisms.

7 Related Work

Our approach to superimposition on the basis of FSTs has been influenced from AHEAD [10]. AHEAD is a model for feature composition that emphasizes language independence of superimposition. Work on AHEAD has claimed that superimposition should be in principle applicable to a wide variety of software artifacts. Work on FEATUREHOUSE has further developed the concept of language-independent superimposition on the basis of FSTs [14]. Composition of UML models is one application scenario that has been analyzed in this paper.

Aspect-oriented modeling (AOM) aims at separating model elements that belong to different concerns. Although features and concerns are not entirely identical concepts [19], decomposing models into features and compose them again on demand is very similar to the aims and procedures of AOM. In AOM, usually, different model fragments are "woven" using certain more or less explicit composition rules. As there is a multitude of different AOM approaches, we use three representative examples to explain the difference to our approach. For example, in the Theme/UML approach, models can be composed by user-defined composition rules that also include to some extent the superimposition by name and type [8]; in the aspectual scenario approach different scenarios are merged and state machines are generated [2]; Jezequel has shown how aspects can be woven into sequence diagrams [26], which is more flexible but also more complex than superimposition. In contrast to these sometimes very different AOM approaches, superimposition on the basis of the FST model is very simple, general, and language-independent, as it can be used to compose so different artifacts like Java, Haskell, and UML models in a uniform way.

Heidenreich et al. propose an approach to enhance modeling languages with composition capabilities based on the meta model [28]. They allow two models to be composed at predefined hooks, which is essentially an interface-based approach. Superimposition is simpler as it merges models by nominal and structural similarities without interfaces, which was sufficient for our case studies. Superimposition is inherently non-invasive whereas the interface-based approach requires a proper preparation.

Several model merge tools, e.g., the Epsilon Merging Language [27] or the ATLAS Model Weaver [3], support diverse kinds of transformations of models. These tools could be used to implement model composition by superimposition. However, superimposition is simpler but also less expressive than other merge tools. As with programming languages, there is a trade-off between simplicity and expressiveness which we find largely unexplored in model transformation.

Boronat et al. present an automated approach for generic model merging from a practical standpoint, providing support for conflict resolution and traceability between software artifacts by using the QVT Relations language [22]. They focus on the definition of an operator *Merge* and apply it to class diagrams integration; other model types are not considered. We have shown that our approach is also applicable to different kinds of models.

FeatureMapper [9] and fmp2rsm [29] are tools with which developers can annotate models with features. Annotations help programmers to overview and understand how individual features influence the structure and behavior of a software system. Annotation is conceptually related to the decomposition of models into features. Annotation and decomposition deserve further investigation since they have complementary strengths and weaknesses [30]. In a recent study, FeatureMapper and FEATUREHOUSE have been used to annotate/decompose entity-relationship-diagrams [31]. However, the focus of this work was on tailoring database schemas and not on model composition.

Feature-oriented model-driven development is an approach that ties feature composition to model-driven development [5]. The core of this approach is a theory based on category theory that relates transformations that stem from feature composition to transformations that stem from model refinement. In their case study, they use the Xak tool [20] to compose state machines written in a domain-specific language, which is related to FEATUREHOUSE but not language-independent.

The package merge mechanism of UML is related to superimposition [32]. It combines the content of two packages. Package merge is used extensively in the UML 2 specification to modularize the definition of the UML 2 metamodel and to define the four compliance levels of UML 2. However, package merge is not applicable to states and roles, and defines many specific composition rules. Our implementation is much simpler and extends to other kinds of models.

Finally, there is a relationship to domain and variability modeling techniques [1,4,17]. While these techniques are used to model the variability of software systems in terms of features and their relationships, we support the derivation of different model variants based on feature composition.

8 Conclusion

We have explored the feasibility and expressiveness of superimposition as a model composition technique. Our analysis and case studies indicate that, even though superimposition is syntax-driven and quite simple, it is indeed expressive enough for the systems we looked at – even in the context of a real-world, industrial case study. We offer a tool that is able to compose UML class, state, and sequence diagrams in the form of XMI documents via superimposition. In further work, we will extend the analysis, the tool, and the case studies in order to support further kinds of UML and non-UML models. Furthermore, we will explore the connection of our metamodel (annotated grammar) to other metamodels for superimposition. Finally, we will experiment with specific (semantics-based) composition rules for individual model elements in order to explore the trade-off between simplicity and expressiveness and to combine them with superimposition eventually.

Acknowledgments

Apel's work was funded partly by the German Research Foundation, project #AP 206/2-1. Trujillo's work was co-supported by the Spanish Ministry of Science & Education under contract TIN2008-06507-C02-02.

References

1. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, CMU (1990)
2. Whittle, J., Araujo, J.: Scenario Modelling with Aspects. IEE Software 151, 157–172 (2004)
3. Jossic, A., et al.: Model Integration with Model Weaving: A Case Study in System Architecture. In: Proc. Int. Conf. Systems Engineering and Modeling, pp. 79–84. IEEE CS, Los Alamitos (2007)
4. Sinnema, M., Deelstra, S.: Classifying Variability Modeling Techniques. Inf. Softw. Technol. 49, 717–739 (2007)
5. Trujillo, S., Batory, D., Díaz, O.: Feature Oriented Model Driven Development: A Case Study for Portlets. In: Proc. Int. Conf. Software Engineering, pp. 44–53. IEEE CS, Los Alamitos (2007)
6. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2002)
7. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. In: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
8. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Addison-Wesley, Reading (2005)
9. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Controlled Visualisations in Software Product Line Engineering. In: Proc. Int. Workshop Visualisation in Software Product Line Eng., Lero, pp. 335–342. University of Limerick (2008)
10. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering 30, 355–371 (2004)
11. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 125–140. Springer, Heidelberg (2005)
12. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proc. Int. Symp. Foundations of Software Eng., pp. 127–136. ACM Press, New York (2004)

13. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 20–35. Springer, Heidelberg (2008)
14. Apel, S., Kästner, C., Lengauer, C.: FeatureHouse: Language-Independent, Automatic Software Composition. In: Proc. Int. Conf. Software Engineering. IEEE CS, Los Alamitos (2009)
15. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. Int. Conf. Software Engineering, pp. 107–119. IEEE CS, Los Alamitos (1999)
16. Bosch, J.: Super-Imposition: A Component Adaptation Technique. Information and Software Technology 41, 257–273 (1999)
17. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
18. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
19. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Software Engineering 34, 162–180 (2008)
20. Anfurrutia, F., Díaz, O., Trujillo, S.: On Refining XML Artifacts. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 473–478. Springer, Heidelberg (2007)
21. Grose, T., Doney, G., Brodsky, S.: Mastering XMI. OMG Press (2002)
22. Boronat, A., Carsí, J., Ramos, I., Letelier, P.: Formal Model Merging Applied to Class Diagram Integration. Electron. Notes Theor. Comput. Sci. 166, 5–26 (2007)
23. Klein, J., Helouet, L., Jezequel, J.: Semantic-Based Weaving of Scenarios. In: Proc. Int. Conf. Aspect-Oriented Software Development, pp. 27–38. ACM Press, New York (2006)
24. Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: An Algebraic View on the Semantics of model Composition. In: Proc. Europ. Conf. Model Driven Architecture – Foundations and Applications, pp. 99–113. Springer, Heidelberg (2007)
25. Czarnecki, K., Pietroszek, K.: Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In: Proc. Int. Conf. Generative Programming and Component Engineering, pp. 211–220. ACM Press, New York (2006)
26. Jezequel, J.M.: Model Driven Design and Aspect Weaving. Software and Systems Modeling 7, 209–218 (2008)
27. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging Models with the Epsilon Merging Language (EML). In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)
28. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Language-Independent Model Modularisation. Trans. Aspect-Oriented Software Development (2009)
29. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
30. Kästner, C., Apel, S.: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: Proc. Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, Dept. of Informatics and Mathematics, University of Passau, pp. 35–40 (2008)
31. Siegmund, N., et al.: Bridging the Gap between Variability in Client Application and Database Schema. In: Proc. Conf. Datenbanksysteme für Business, Technologie und Web, Gesellschaft für Informatik, pp. 297–306 (2009)
32. Group, O.M.: Unified Modeling Language: Superstructure, Version 2.1.1 (2007)

Efficient Model Transformations by Combining Pattern Matching Strategies

Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
{bergmann,ahorvath,rath,varro}@mit.bme.hu

Abstract. Recent advances in graph pattern matching techniques have demonstrated at various tool contests that graph transformation tools can scale up to handle very large models in model transformation problems. In case of *local-search* based techniques, pattern matching is driven by a search plan, which provides an optimal ordering for traversing and matching nodes and edges of a graph pattern. In case of *incremental pattern matching*, matches of a pattern are explicitly stored and incrementally maintained upon model manipulation, which frequently provides significant speed-up but with increased memory consumption. In the current paper, we present a *hybrid pattern matching* approach, which is able to combine local-search and incremental techniques on a per-pattern basis. Based upon experimental evaluation, we identify scenarios when such combination is highly beneficial, and provide guidelines for transformation designers for optimal selection of pattern matching strategy.

1 Introduction

Model transformations play a crucial role in modern model-driven system engineering, an application domain where transformations need to handle large, industrial models in a short amount of time. *Graph transformation* (GT) [1] based tools have been frequently used for capturing and executing complex transformations. In GT tools, *graph patterns* capture structural conditions and type constraints in a compact visual way. During transformation time, these conditions are evaluated during *graph pattern matching*, which aims to derive one or all matches of a given pattern to execute a transformation rule.

Empirical evidence reported at recent tool contests [2,3] proved that GT tools scale up for transforming very large models, thanks to highly sophisticated, *local-search based graph pattern matching* (LS) algorithms proposed in transformation tools such as GrGEN.NET [4], FUJABA [5], and VIATRA2 [6]. In all these approaches, pattern matching is driven by a search plan, which provides an optimal ordering for traversing and matching nodes and edges of a graph pattern.

As an alternative, incremental pattern matching (INC) approaches [7,8,9,10,11] have recently become a hot topic in the model transformation community. The core idea is to improve the execution time of the time-consuming pattern matching phase by additional memory consumption. Essentially, the (partial) matches of graph patterns are stored explicitly, and these match sets are updated incrementally in accordance with elementary model changes. While model manipulation becomes slightly more complex, all matches of a graph pattern can be retrieved in constant time in exchange by eliminating the need for recomputing existing matches.

Initial benchmarking [12] has shown that in many scenarios, the incremental pattern matching approach (as implemented in the VIATRA2 framework) leads to orders-of-magnitude increases in speed. However, an important implication of caching match sets is increased memory consumption, which needs to be taken into account when scaling up to large models. Unfortunately, in many practical applications of model transformations, available memory is frequently constrained (e.g. when they are executed on average desktop computers and not on high performance servers).

In the current paper, we propose a hybrid pattern matching approach which enables the transformation designer to combine local search-based and incremental pattern matching to adapt to memory constraints. At design-time, transformation engineers may select whether a graph pattern should be matched using the LS or the INC strategy separately for each pattern. Moreover, based upon runtime monitoring, the execution engine may automatically switch from incremental pattern matching to local-search based technique when a certain memory limit has been reached.

Additionally, we present experiments to demonstrate that the incremental strategy is not always the best choice for pattern matching: we highlight scenarios using a performance benchmark of model transformations (object-relational mapping) where a combination of INC with LS significantly outperforms the plain INC-only and LS-only versions. By the analyzing results of our case study, we provide a list of various factors (metrics) which we experienced to have significant factor on performance, and give hints to transformation designers when a graph pattern should be matched using an INC or an LS strategy in each case.

The rest of the paper is structured as follows. Section 2 briefly introduces graph patterns and graph transformation rules (as available in the VIATRA2 transformation language). It also describes the object-relational mapping used as a performance benchmark throughout the paper. As related work, we highlight main characteristics of the local search-based and incremental pattern matcher implementations. In Section 3, we present scenarios to highlight when a hybrid pattern matching strategy provides significant performance advantage in typical model transformations. We present metrics to optimally select LS or INC strategies for patterns at design time. Moreover, we present an adaptive runtime technique to switch to LS strategy when memory is low (Section 4). Finally, Section 5 concludes the paper.

2 Background

2.1 Graph Patterns and Transformation

Graph patterns are frequently considered as the atomic units of model transformations [13]. They represent conditions (or constraints) that have to be fulfilled by a part of the instance model. A basic graph pattern consists of (i) *structural constraints* prescribing the existence of nodes and edges of a given type, and (ii) *containment constraints* specifying containment relation between nodes of graph patterns. A *negative application condition* (NAC), defined by a negative subpattern, prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match.

Graph transformation (GT) [1] provides a high-level rule and pattern-based manipulation language for graph models. Graph transformation rules can be specified by using a left-hand side – LHS (or precondition) pattern determining the applicability of the rule, and a right-hand side – RHS (postcondition) pattern which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged.

In the following, we use the language of the VIATRA2 framework [13] for demonstration purposes, which also provides additional *control structures* (such as rule invocation, variable declaration, sequencing operator, iterative and simultaneous execution etc.) for assembling complex transformations defined by abstract state machines (ASM) [14].

Example 1. The graphical representation of an example graph transformation rule, along with the VIATRA2 textual representation of the LHS, is depicted on Figure 1. Informally, the meaning of the *schemaRule* is to map a package *P* contained by the container *Con* to a schema *S*, unless the mapping has already been performed. Elements labeled with *new* are the ones created by the rule; elements labeled with *neg* constitute the single NAC of the rule[1]. The NAC is used to exclude the cases when the mapping has been performed; it applies if there is an edge *R* of type schemaRef between the nodes *P* and *SN*.

How this rule is invoked in VIATRA2 is captured by Lst. 1. The *forall* rule is used to find all substitutions (matches) for variables defined in its head (*P*), which satisfies the LHS of the *schemaRule* patttern, and then executes the model manipulation operations for each substitution separately followed by the *println* rule. Note that the *Container* variable will have to be defined prior to the execution of the *forall* rule as it is assumed as an input parameter for the *schemaRule*.

```
gtrule schemaRule(P,Con) = {
  package(P) below Con;
  neg pattern mapped(P, SN, RN) = {
    package(P);
    schema(SN);
    package.schemaRef(RN, P, SN);
  }
 new schema(SN);
 new package.schemaRef(RN, P, SN);
}
```

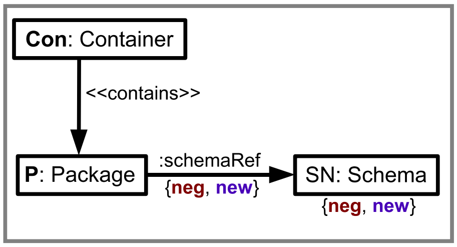


Fig. 1. GT rule for unmapped packages below a container

```
...
 //invoking the schemaRule GT Rule with Container as a bound input parameter
 forall P with apply schemaRule(P,Container) do
  //prints the name of the package processed to the console
  println("SchemaRule: p="+ name(P));
```

Listing 1. VIATRA2 source code for invoking the schemaRule

[1] We do not display rules with multiple NACs or with deletion action in this paper.

2.2 Case Study

Transformation overview. The (simplified) Object-to-Relational schema mapping (ORM) case study was proposed as a performance benchmark of model synchronization transformations in [15, 12]. The aim of the transformation is to produce corresponding relational database schemas from UML class diagrams according to the following mapping rules:

- First, a relational schema is created for the specified package below a given container by *schemaRule* (Fig. 1). Transitive containment is represented by an edge tagged as “contains”.
- Then classes in the package are mapped into tables in the corresponding schema, each with an id column as a primary key (*classRule*).
- Each association in the package is mapped into a table in the corresponding schema with a primary key column (*associationRule*);
- Each association end in the association is mapped into a foreign key of the corresponding table pointing at the table generated from the class that the association end points to (*assocEndRule*);
- Attributes in the class are mapped into columns of the corresponding table (see *attributeRule* in Fig. 3).

In incremental synchronization, to avoid rebuilding target models in each pass, a *reference model* (also known as trace / correspondence model) is used to establish a mapping between source and corresponding target model elements (Fig. 2) and to trace changes in them. In this context, the reference model is often referenced in NACs to identify elements in the source model that have not yet been mapped into the target model. In the current paper, we restrict our investigations to one-way synchronization.

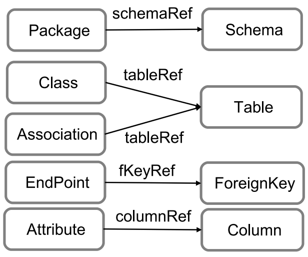


Fig. 2. Reference metamodel

Transformation scenario. In the original benchmark example [12], the source model consists of two packages, both containing a (generated) set of classes with attributes.

```
gtrule attributeRule(C, A, T) = {
  class(C);
  class.attribute(A);
  class.cl2attrs(CF1, C, A);
  table(T);
  class.tableRef(R1, C, T);
  neg pattern mapped(A, ColN, RN) = {
    class.attribute(A);
    column(ColN);
    class.attribute.colRef(RN, A, ColN);
  }
 new column(ColN);
 new class.attribute.colRef(RN, A, ColN);
}
```

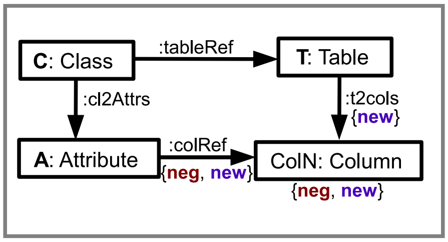


Fig. 3. GT rule for unmapped attributes

```
pattern detectGeneralization(Sub,Sup) = {
  general(Gen);
  class(Sup);
  general.parent(PE,Gen,Sup);
  class(Sub);
  general.child(CE,Gen,Sub);
}
```

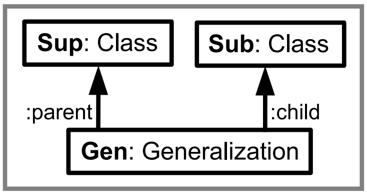


Fig. 4. Graph pattern checking for generalizations

First, (i) the *primary* package will be mapped into a relational schema to create initial mappings. Then, the *source models are modified*, and, in an additional pass, (ii) the system has to *synchronize the changes to the target model* (i.e. find the changes in the source and alter the target accordingly). This scenario is now extended as follows.

Check phase. First, we check as a precondition of the transformation that no generalization exists in the source UML model to ensure the applicability of the transformation. It is captured by a corresponding simple graph pattern (*detectGeneralization* in Fig. 4). We intentionally omitted support for inheritance from the model transformation itself to analyze a case where a transformation has to perform a preliminary applicability check; the practical consequences of this choice will be assessed later in Sec. 3.2.

Initial transformation phase. When there are no generalizations, the primary package is mapped into a relational schema by the transformation program.

Refactoring phase. A refactoring operation modifies the package hierarchy of the source model, namely the secondary package is moved inside the primary package.

Synchronization phase. Afterwards, synchronization propagates these changes into the target relational model so that it holds the mapping of the (changed) primary package once again. This involves creating the tables for classes that stem from the secondary package, and creating columns for attributes of these classes.

2.3 Pattern Matching Strategies and Related Work

Pattern matching plays a key role in the efficient execution of all model transformation engines. In case of graph transformation based approaches, the goal is to find the occurrences of a graph pattern, which contains structural as well as type constraints on model elements. During pattern matching, each variable of a graph pattern is bound to a node in the model such that this matching (binding) is consistent with edge labels, and source and target nodes of the model.

Most model transformation approaches (e.g. [4, 5, 16, 17] and many more) usually rely on a *local search based pattern matching* (LS) that starts the matching process from a single node and extends it step-by-step by neighboring nodes and edges. Informally, a search plan [18, 6] defines an ordering of pattern nodes, in which they are bound to objects of the instance model during pattern matching. With efficient search plans ([4, 19]), LS strategies can produce good runtime performance with a relatively small memory footprint, and low update complexity. Other approaches [20, 21] use constraint satisfaction techniques for matching graph patterns.

As an alternate approach, *incremental pattern matching* (INC) [7,8,10,11] relies on a *cache* in which the matches of a pattern are stored explicitly. The match set is readily available from the cache at any time without searching, and the cache is incrementally updated whenever changes are made to the model. As pattern occurrences are stored, they can be retrieved in constant time – excluding the linear cost induced by the size of the result set itself –, making pattern matching extremely fast. The trade-off is increased memory consumption, and increased update costs (required to continuously maintain the stored match set caches.

In the current paper, our goal is to investigate if (and when) the combination of pattern matching strategies within a transformation (referred to as *hybrid pattern matching*) can provide better runtime performance, especially, with constraints on available resources (such as memory consumption). For our investigations, we use the VIATRA2 framework, which supports both pattern matching engine strategies and allows to specify the use of INC or LS strategy separately for each graph pattern.

There are cases where the use of either the incremental or the local search based pattern matching approach is significantly more efficient than the other. We argue that many transformations could benefit even more from combining these two approaches, by using different pattern matcher engines for different graph patterns. As a conceptual analogy for our current work, recent research in expert systems [22] demonstrated that an integration between two different incremental strategies can be advantageous.

3 Motivating Scenarios for Hybrid Pattern Matching

Recent benchmarks evaluations [12] and tool contests [3] in the graph transformation community have shown that INC can easily be orders of magnitude faster than (most) LS approaches for certain problem classes. This section identifies three scenarios where, on the other hand, LS has a clear advantage, as demonstrated by our experiments[2]. For each scenario, we identify a hybrid pattern matching approach where some patterns and transformations should use LS, while the rest of the transformation relies upon INC to obtain a better performance than the two extremes (LS-only or INC-only).

3.1 Scenario: Match Set Cache Does Not Fit into Memory Limit

This scenario demonstrates that the high memory consumption of incrementally maintained caches can be a bottleneck of INC. By choosing LS for patterns that are memory-intensive (i.e. with many matches) but not time-critical, the high memory consumption can be greatly reduced, while still retaining the short execution time comparable to INC.

Our experiments were performed on the *Transform phase* of the ORM case study (see section 2.2), by measuring the heap commit size of the Java VM. In the followings, we model a frequently occurring development scenario. As the transformation designer is typically working with small toy models, scaling up to large model sizes might lead to unexpected results. For instance, while a toy model with 10 classes and 250 attributes and the corresponding INC cache easily fits in a few megabytes, a memory usage of

[2] Measurement environment: Intel Core Duo t2400@1,83 GHz processor, 3 GB RAM, Windows XP SP3, Sun HotSpot Java 1.6.0_02 and VIATRA2 Release 3 build 2009.02.03.

128MBs can be reached by increasing the model to 575 classes and 14375 attributes, as shown on Table 1. With a memory limit of 128M, as the match set cache expands rapidly, the JVM begins to trash due to memory starvation shortly after the transformation is started. This leads to significant slowdown (to 21 seconds), and may even result in a failed execution because of heap exhaustion. If the amount of memory is suitably large (i.e. 1GB in our case), execution is very fast (4.6 seconds). LS is not an alternative here: while the memory consumption of the caches is spared, the execution time for this model size is very long (avg. 184 seconds).

Closely observing this transformation, we may identify LHS pattern *attributeRule* (see Figure 3) and its embedded negative application condition as patterns with high number of occurrences. By sacrificing execution time (runs in 5.0s with a 1G heap), we marked this pattern to be matched by the LS engine, despite using INC for the rest of the transformation. This reduced memory consumption to 105M, and allowed the transformation to run with approximately the same execution time (5.6s) even with a memory limit of 128M. Therefore the hybrid approach has the potential to efficiently scale up to higher model sizes given the same memory constraints.

Table 1. Match Set Memory and Performance

PM Strategy	Memory limit [MB]	Used heap [MB]	Transform phase time [ms]
LS	128	99	183729
INC	128	128	21057
INC	1024	128	4639
Hybrid	128	105	5573

3.2 Scenario: Construction Time Penalty

This scenario emphasises that the time required to initialize the incrementally maintained caches might itself be too expensive. The construction time of the caches is not less than the time required to find all occurrences of the pattern, since the match set is directly available from this cache. If the transformation needs to find only one (or few) of many pattern occurrences altogether, there is no need for LS to continue the search and retrieve the entire match set, therefore it can be significantly faster than INC. This phenomenon only applies if the pattern is efficiently matchable by LS, unlike large patterns with high combinatorial complexity.

This behaviour was observed in the *Check phase* of the ORM case study (see section 2.2). We measured the time it takes to find an arbitrary generalization edge if all 2500 generated classes inherit a common superclass, which is a single-occurrence query of a very simple graph pattern (see Figure 4) consisting of a single edge. The measurements show (in Table 2) that constructing the cache took 0.14s on average, while INC would gain only about 16 ms time (too small to be more accurately measured) compared to LS with each further query (if there were any). To complement these results, we also took the measurement on a source model without generalization, for which the transformation could be performed; we found that, in accordance with expectations, LS has no significant advantage in this case: constructing the cache took 16 ms, while LS needed 36 ms to complete the query (both of them too small to be accurately measured).

Table 2. Construction Time Performance

PM Strategy	Used heap [MB]	Cache construction time [ms]	Further queries [ms]
with generalization edges			
LS	152	-	16
INC	159	143	0
without generalization edges			
LS	147	-	36
INC	149	16	0

3.3 Scenario: Expensive Model Updates

This scenario happens if there is heavy model manipulation between infrequent pattern queries. In this case, the time overhead imposed on model manipulation by INC may outweigh its benefits. The cost of incrementally maintaining the match set caches for a long period of time with frequent model updates may be larger than the cost of applying LS and calculating the match set from scratch at each pattern query. In other terms, it may be superfluous to continuously maintain the match sets if they are not frequently used for model queries. This cost can be avoided by not using INC, or only using it for a limited number of patterns.

Expensive update overhead is observable in the *Refactoring phase* of the ORM case study (see section 2.2). We measured the time it takes to move a package in the source model to a different package, while the INC maintains the caches of patterns that represent the location of classes in the namespace hierarchy of packages, classes and attributes (Table 3). The transitive containment is a model feature of high combinatorial complexity, and moving a high-level element will cause drastic changes in this relationship, thereby forcing the INC to perform intensive cache updates. The measurements have shown that the cost of the single move operation can be as high as 2.1 seconds with INC. Using pure LS was not a feasible solution either, as the *Synchronization phase* did not terminate within half an hour. A hybrid pattern matcher assignment solved these problems: patterns using the transitive containment (see Figure 1) were matched by LS and the rest by INC, resulting in a fast move operation and an execution time of 14.5 ms for the entire *Refactoring phase*. These measurements were taken with both the primary and secondary packages consisting of 1000 classes and 25000 attributes.

Table 3. Model Update Performance

PM Strategy	Used heap [MB]	Refactoring phase time [ms]	Synchronization phase time [ms]
LS	-	0	>2000000
INC	493	2109	13386
Hybrid	298	0	13570

3.4 Overall Performance on the Entire Case Study

Finally, we compare the overall performance of the three approaches on all three steps of the case study combined. Measurements were taken for various source model sizes,

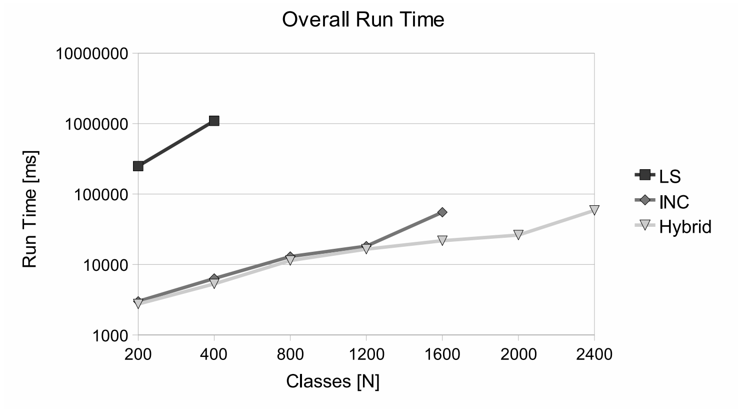


Fig. 5. Overall Execution Time

scaling up until the transformation became too slow (LS) or did not fit into memory (INC, hybrid). Figure 5 indicates the total execution time versus the number of classes in the primary source package. For these measurements, the number of classes in the secondary package (N/4) was always one quarter of the number of classes initially in the primary package (N), and each class still had 25 attributes (25N, 25N/4); thus the largest case (N=2400) consisted of 2400 classes and 60000 attributes in the primary package, 600 classes and 15000 attributes in the secondary package, i.e. more than 150 000 source model elements altogether including edges. As the figure shows, INC scales up higher than LS, but the hybrid approach is even more efficient (note that due to overhead, the advantage becomes visible for large models when $N >= 1200$).

4 Towards Intelligent Selection of Matching Strategies

In this section, we first *identify various factors* (qualitative metrics) which help transformation designers decide when a certain pattern matching strategy (LS or INC) would be beneficial (Sec. 4.1). Then, in Sec. 4.2, we discuss how an *adaptive run-time behaviour* can be obtained by monitoring relevant metrics, and switching from one strategy to the other at runtime. Compared to existing adaptive pattern matching solutions [19,4], the main novelty of this approach lies in the fact that we are able to *automatically switch between two entirely different pattern matching strategies* to increase performance. The high-level workflow of these techniques is illustrated in Fig. 6.

As identified in Sec. 3, several factors may influence the behaviour of the pattern matching algorithms. *Static factors* like (i) *static attributes of graph patterns* (e.g. pattern size, fan-out, structural complexity) and (ii) *control structures* of model transformations (e.g. forall, iterate) determine operative characteristics which, in combination with the characteristics with the different pattern matcher strategies, greatly influence the cost of pattern matching.

In contrast, *dynamic factors* change in-between transformation runs on the same system, and also with different target execution platforms: (iii) *model-specific graph*

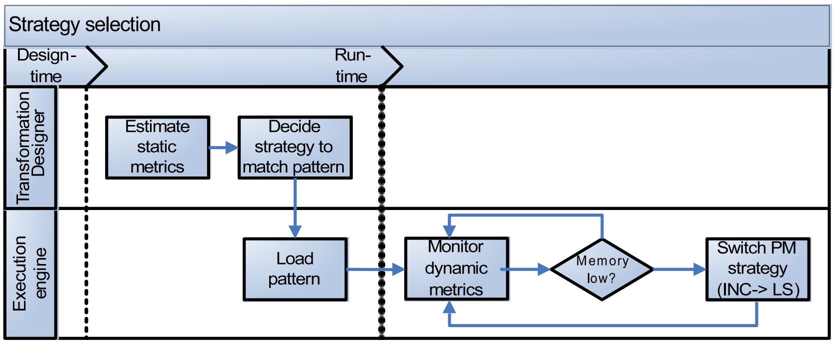


Fig. 6. Selecting pattern matching strategies at design-time and runtime

characteristics like qualitative attributes related to structure (e.g. average fan-out) and quantitative parameters related to model size (e.g. total number of model elements) may change as the transformation is changing the underlying model. Moreover, (iv) *memory limitations* impose external constraints which are related to the execution environment.

4.1 Factors for Design-Time Selection of Matching Strategies

In the VIATRA2 framework, transformation designer can fine-tune the performance of graph pattern matching by prefixing a graph pattern with `@incremental` or `@localsearch` annotations to select the designated pattern matching strategy.

Based on our previous experience with performance benchmark transformations [12] and practical model transformations of large complexity [23], we identified the following factors to be important for transformation designers to choose between LS and INC strategies:

(i) *Graph pattern static attributes*

- *number of graph patterns* in a transformation program has a huge impact on the memory consumption. The cache size of the pattern increases memory consumption when matched by INC strategy.
- *pattern size*: in practical applications, we experienced that the number of matches gradually decrease as the pattern to be matched becomes more and more complex (having more and more elements). This contradicts the intuition that larger patterns will have more matches due to more combinatorical possibilities. Although this combinatorical increase may hold for smaller patterns, it is overwhelmed by the scarcity due to restrictiveness of larger patterns in many practical scenarios. As a result, large patterns should be preferably matched by INC.
- *containment hierarchy constraints*, especially transitive containment, may significantly increase the memory consumption of incremental pattern matching due to fact that all containment relation between model elements have to be *cached* and incrementally updated. A good compromise could be to decompose the pattern and match only the containment constrained part with the LS

engine while leaving INC strategy for the rest. Another solution would be to refactor patterns (and possibly the model also) so that they use explicit graph edges instead of relying on the implicit containment hierarchy.

(ii) *Control structures*
- *parameter passing* is using the result of rules or patterns as an input of other rules or patterns. This technique increases efficiency in LS as search operations are much more efficient if one or more pattern variables are bound, i.e. their values are known at time of the query. INC performance is not affected.
- *usage frequency* of patterns is relevant, since the more often a pattern is used, the more advantage INC has. Frequently used patterns can be identified by static analysis of the transformation code, e.g. by marking patterns that are used from within a loop. Trace analysis can yield more valuable estimates, if typical example inputs are available, by executing the transformation on these inputs and counting the times each pattern is accessed.
- *model update cost*: if program code analysis can reveal that model element types belonging to a certain pattern are rarely (or never) manipulated, the model manipulation costs imposed by INC can be neglected.

(iii) *Model dependent pattern characteristics*
- *node type complexity*, a rough upper bound on the number of potential matches can be obtained as the product of the cardinalities (number of model instances) of the types of each node in the graph pattern. This estimate is, of course, accurate as there are also edges in the pattern to constrain the possible combinations of nodes. However, high complexity may result in high memory consumption for INC, and long search operations for LS.
- *model statistics* generally extend graph pattern static attributes to the entire instance model the transformation is working on. A well-known practical statistics on pattern complexity is the *search space tree cost*, that has already been used to adaptively select the search plan for LS-based matchers [19]. It uses model statistics to assess the branching factors (node type complexity) during the search process. Other important factors like fan-out, hierarchy depth and model symmetries can also effectively make the estimation of match set sizes and time complexity of the pattern matching more precise.

By evaluating these (qualitative) metrics on the ORM case study described in Sec. 2.2, the observed behaviour in Scenarios 3.1–3.3 can be explained in more detail.

– In Sec. 3.1, we have identified the cause of the performance bottleneck to the *attributeRule* graph pattern with large match set. Since this pattern is used to filter for Attributes which have not yet been mapped to a Table column, it can be expected to have an initially large match set for class models with a large number of attributes. The match set size can be estimated a-priori by looking at *instance count* numbers for the Attribute type, or, by simply considering the general type composition characteristics of models the transformation is to be executed on.
– Sec. 3.2 demonstrated the usage of a simple pattern for structural checking (i.e. executing only once). This case corresponds to low pattern complexity and low usage count which, especially when combined with a potentially high match count, indicates a good candidate for switching to LS.

- Finally, Sec. 3.3 uses a pattern with a transitive containment constraint which, when used for synchronization after a model move high in the containment hierarchy, caused a drastic overhead for the incremental pattern matcher. As the resolution suggests, such patterns should generally be matched with LS.

4.2 Adaptive Runtime Optimization

Dynamic factors like memory consumption can quite easily change in-between transformation runs (even on the same system), especially using INC pattern matching, leading to performance degradation or insufficient memory. The current section focuses on an adaptive approach that can intervene in the predefined matching strategy in order to adapt to the altered environment.

In accordance with the general strategy described in Sec. 3, the adaptive engine generally prefers using the incremental pattern matcher for all graph patterns. When shortage of available memory is detected, pattern match set cache structures are gradually abandoned. For constructing such an adaptive approach monitoring, the following parameters are actually considered:

- During the execution of a VIATRA2 transformation the memory consumption is directly observable through the Java Virtual Machine (JVM), which provides a straightforward way for *monitoring available memory*.
- Simple *model space statistics* (e.g. the total number of model elements) are automatically registered by the VIATRA2 engine, along with sizes of match sets available from the incremental pattern matcher that can also be used as a model-specific indicator for actual memory consumption and to dynamically detect situations where run-time adaptive matching selection strategy switching is needed.

Note that telemetry registration does have some overhead at run-time (especially in the case of heap monitoring since several garbage collection runs need to be executed for reliable heap data), however this overhead is negligible for long-running transformations.

For the actual strategy the priority order for the cache removal is determined by the *largest-first* principle, where the pattern match cache structure with the largest overall memory footprint is selected for removal resulting, that the forthcoming pattern match operation requested for the corresponding pattern will always be executed by the LS-based pattern matcher leading to a smaller memory consumption. In our case, memory shortage is detected when the available heap memory is less than 15%, which initiates dropping PM caches and switching to LS strategy.

In order to evaluate the efficiency and impact of this approach, we ran the benchmark experiment described in Sec. 3.1 with the adaptive implementation. The results for this measurement were obtained in a different software environment: we used the 64-bit version of IcedTea 1.3.1 as a JVM (hence the larger memory consumption figures). Execution times can be observed in Table 4.

Unsurprisingly, the execution time of the hybrid adaptive approach is between the fastest INC, the static hybrid approaches and a pure LS run. Note that memory was constrained for hybrid runs, marked with *; with memory constraints, INC would not run successfully in this case.

Table 4. Match Set Memory and Performance of the Adaptive Hybrid Strategy

PM strategy	Used heap [MB]	Transform phase time [ms]
LS	201	77054
INC	353	13693
Static hybrid	220*	10958
Adaptive hybrid	235*	35716

Overall, this technique prevents the transformation engine from trashing due to memory starvation. However, the largest match set caches may not be the best choice for abandonment when optimizing for the shortest possible execution time. Therefore the presented technique is theoretically sub-optimal. A straightforward approach for future optimization is adjusting the priority order based on static analysis of the transformation program.

5 Conclusion and Future Work

Practical experience has shown that performance optimization is an important part of building powerful model transformations in a model-driven development process. First, as models are increasing in size and complexity, transformations need to be able to transform them efficiently. Secondly, as transformations are becoming *hidden* (e.g. embedded in a design tool), they should execute seamlessly - quickly and using as little resources as possible.

In this paper, we presented a *hybrid pattern matching* approach, which provides smart selection from two entirely different matching strategies (namely, the local search-based and incremental pattern matching) to improve overall performance.

Based on experience with complex applications of model transformations (e.g. [23]), we selected three scenarios for the investigation. Based on our experimental analysis, we argue that many practical transformations may significantly benefit from a hybrid pattern matching approach with properly selected matching strategies for the patterns. We gave conceptual guidelines on manual optimization based upon various metrics in Sec. 4. Additionally, as an initial contribution towards automatic optimization, we presented an adaptive approach switches pattern matching strategies when memory is running low.

However, we also recognize that the ultimate goal for optimizing model transformation performance is to enable the user to concentrate only on functionality and the software tool should select the optimal pattern matching strategy. In order to provide semi-automatic aids to the transformation designer for code optimization, and to develop a more optimal method for adaptive strategy switching, several well-known approaches can be adapted in the future.

– *Pattern analysis* may be used to classify graph patterns according to complexity, size, and complex cost metrics (as mentioned in Sec. 4.1) statically. While such techniques are currently used internally in our LS implementation, direct user interface feedback is needed to expose relevant data to the transformation designer.

– *Program analysis* aims to identify patterns and model manipulation steps that are frequently used, rarely used, or unused for a period of time by analyzing the transformation program, without actually running it.
– *Trace analysis* improves this knowledge of transformation behaviour by actually running the program on one or more provided typical models and gathering statistics on the type and amount of executed pattern queries and model manipulations.
– *Quantitative model analysis* is a highly promising approach to estimate the match set cardinality of graph patterns based on statistics of the model (without actually running the pattern matching algorithm).

As a main direction for future work, we plan to implement a framework with high-level support for these static analysis techniques, to find answers for open questions outlined in Sec. 4.1 (e.g. the limit in pattern size and pattern usage frequency for a given transformation where the break-even point for INC and LS occurs).

Additionally, we plan to investigate ways to achieve tighter integration between the two pattern matching engines. This will allow different strategies to be responsible for matching different subpatterns within the same pattern.

While the direct contributions of the paper are dedicated to graph transformation-based approaches of model transformations, we believe that the conceptual foundations are, in fact, adaptable to other transformation techniques. For instance, similar investigations can be carried out in the future to assess when an OCL constraint should be evaluated incrementally, and when an evaluation should be initiated from scratch.

References

1. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook on Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
2. The AGTIVE Tool Contest: official website (2007), `http://www.informatik.uni-marburg.de/~swt/agtive-contest`
3. GraBaTs - Graph-Based Tools: The Contest: official website (2008), `http://www.fots.ua.ac.be/events/grabats2008/`
4. Geiss, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006)
5. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: The 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland. ACM Press, New York (2000)
6. Varró, G., Horváth, Á., Varró, D.: Recursive Graph Pattern Matching With Magic Sets and Global Search Plans. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088. Springer, Heidelberg (2008)
7. Varró, G., Varró, D., Schürr, A.: Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In: Karsai, G., Taentzer, G. (eds.) Graph and Model Transformation (GraMoT 2006). Electronic Communications of the EASST, vol. 4. EASST (2006)
8. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT 2008, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (2008)

9. Matzner, A., Minas, M., Schulte, A.: Efficient Graph Matching with Application to Cognitive Automation. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088. Springer, Heidelberg (2007)
10. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
11. Mészáros, T., Madari, I., Mezei, G.: VMTS AntWorld submission. In: GraBaTs - 4th International Workshop on Graph-Based Tools: The Contest (September 2008)
12. Bergmann, G., Horvath, A., Ráth, I., Varr, D.: A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214. Springer, Heidelberg (2008)
13. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming 68(3), 214–234 (2007)
14. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
15. Varró, G., Schürr, A., Varró, D.: Benchmarking for Graph Transformation. Technical Report TUB-TR-05-EE17, Budapest University of Technology and Economics (March 2005), http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf
16. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In: [1], pp. 487–550. World Scientific, Singapore (1999)
17. ATLAS Group: The ATLAS Transformation Language, http://www.eclipse.org/gmt
18. Zündorf, A.: Graph Pattern Matching in PROGRES. In: Selected papers from the 5th International Workshop on Graph Gramars and Their Application to Computer Science, London, UK, pp. 454–468. Springer, Heidelberg (1996)
19. Varró, G., Varró, D., Friedl, K.: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In: Karsai, G., Taentzer, G. (eds.) Proc. of Int. Workshop on Graph and Model Transformation (GraMoT 2005), Tallinn, Estonia. ENTCS, vol. 152, pp. 191–205. Elsevier, Amsterdam (2005)
20. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 238–252. Springer, Heidelberg (2000)
21. El-Boussaidi, G., Mili, H.: Detecting patterns of poor design solutions using constraint propagation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 189–203. Springer, Heidelberg (2008)
22. Wright, I., Marshall, J.: The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. International Journal of Intelligent Games and Simulation 2(1), 36–48 (2003)
23. Kovács, M., Lollini, P., Majzik, I., Bondavalli, A.: An Integrated Framework for the Dependability Evaluation of Distributed Mobile Applications. In: Proc. Int. Workshop on Software Engineering for Resilient Systems (SERENE 2008), Newcastle upon Tyne, UK, November 17-19, pp. 29–38 (2008)

Managing Dependent Changes in Coupled Evolution[*]

Antonio Cicchetti[1], Davide Di Ruscio[2], and Alfonso Pierantonio[2]

[1] School of Innovation, Design and Engineering
Mälardalen University,
SE-721 23, Västerås, Sweden
antonio.cicchetti@mdh.se

[2] Dipartimento di Informatica
Università degli Studi dell'Aquila
Via Vetoio, Coppito I-67010, L'Aquila, Italy
{diruscio,alfonso}@di.univaq.it

Abstract. In Model-Driven Engineering models and metamodels are not preserved from the evolutionary pressure which inevitably affects almost any artefacts. Moreover, the coupling between models and metamodels implies that when a metamodel undergoes a modification, the conforming models require to be accordingly co-adapted. One of the main obstacles to the complete automation of the adaptation process is represented by the dependencies which occur among the different kinds of modifications. The paper illustrates a dependency analysis, classifies such dependencies, and proposes a metamodeling language driven resolution which is independent from the evolving metamodel and its underlying semantics. The resolution enables a decomposition and consequent scheduling of the adaptation steps allowing the full automation of the process.

1 Introduction

Model Driven Engineering (MDE) [1] is increasingly gaining acceptance as a mean to leverage abstraction and render business logic resilient to technological changes. Coordinated collections of models and modelling languages are used to describe software systems on different abstraction layers and from different perspectives [2]. In general, domains are analysed and engineered by means of a *metamodel*, i.e. a coherent set of interrelated concepts. A model is said to *conform* to a metamodel, or in other words it is expressed by the concepts encoded in the metamodel, constraints are expressed at the metalevel, and model transformation occurs when a source model is modified to produce a target model.

In a model-centric vision of software-development, models and metamodels are not preserved from the evolutionary pressure which inevitably affects almost any artefacts involved in the process [3]. Moreover, the coupling between models and metamodels implies that when a metamodel undergoes a modification, the conforming models require to be accordingly *co-adapted*[1] not to let them become invalid. This adaptation

[*] Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n° 214898.

[1] The terms co-adaptation, co-evolution, and coupled evolution will be at some extent used as synonyms throughout the paper.

process is difficult, error-prone and can give place to inconsistencies between the metamodel and the related artefacts, if not supported by any automation. Such an issue becomes even more relevant when dealing with enterprise applications, since in general system models encompass a large population of instances which need to be appropriately adapted, hence inconsistencies can possibly lead to irremediable information erosion [4]. The management of coupled evolution is intrinsically complex and requires the capability of *a) differencing*, i.e. determining the differences between two versions of the same metamodel and *b) adaptation*, that is a transformational process able to partly or fully automatize the adaptation of the models according to the modifications detected in the previous step. Recently, these aspects have been investigated by several works, while some focused on the problem of metamodel matching (e.g., [5]), most of them concentrated on the adaptation by either assuming that change traces, for instance, are somehow available or addressing only atomic modifications (e.g., [4,6,7]), see Sect. 2.1 for a detailed discussion. Unfortunately, supposing the availability of predefined information about changes and assuming only atomic operations is not always practicable, because metamodels usually evolve in a complex way without keeping track of the applied changes.

This paper proposes a transformational approach to co-adaptation which is agnostic of the differencing method and considers complex modifications of metamodels, in contrast with current approaches [4,6,7]. As shown in [8], the adaptation is defined as the parallel composition of two different transformations which are automatically derived from the *breaking resolvable*, and *breaking unresolvable* changes. Unfortunately, the occurrence of dependencies between these two kind of changes compromises the parallel independence of the generated transformations, and thus the complete automation of the co-adaptation. This work enhances the work in [8] by proposing a dependency analysis which underpins a resolution strategy allowing the correct scheduling of the adaptation steps. All the metamodel change dependencies have been considered and for each of them a resolution schema is proposed enabling the complete automation of the adaptation. Interestingly, the technique is independent of the metamodel and its underlying semantics, since it relies only on the definition of the metamodeling language.

The structure of the paper is as follows. In Sect. 2 a discussion about the related work and the background is presented. Next section analyzes the metamodel change dependencies and discusses the countermeasures to adopt in order to resolve them. Finally some conclusions are drawn.

2 Metamodel Evolution and Model Co-evolution

Metamodels are expected to evolve during their life-cycle, thus causing possible problems to existing models which conform to the old version of the metamodel and do not conform to the new version anymore. A possible solution is the adoption of mechanisms of model co-evolution, i.e. models are migrated in new instances according to the changes of the corresponding metamodel. In the following, related works are illustrated to give an overall view of the problem, current solutions, and the issues which are still open.

2.1 Related Work

The problem of co-evolution presents intrinsic difficulties. In [7] the authors introduce a new language, COPE, to support the adaptation of models with respect to metamodel updates. However, the language is mainly exploited to provide helpers in instance co-adaptations and not to introduce a generative approach based on metamodel variations. In [4,6,9] the authors try to improve the degree of automation, by considering all the possibile metamodel manipulations and distinguishing them with respect to the effects they have on the existing instances. In particular, metamodel changes are classified in *(i) non-breaking changes* that do not break the conformance of models once the corresponding metamodel has been modified, *(ii) breaking and resolvable changes* which break the conformance of models even though they can be automatically co-adapted, and *(iii) breaking and unresolvable changes* that break the conformance of models which can not be automatically co-evolved and user intervention is required. Such a categorization suggests to support model co-evolution by separating the various forms of metamodel revisions and then by adopting the appropriate countermeasures. For instance, in [4] metamodel evolutions are specified by QVT relations, while co-adaptations are defined in terms of QVT transformations when resolvable changes occur. The main limitations are that co-adapting transformations are not automatically obtained from metamodel modifications and unresolvable changes are not given explicit support. Moreover, using relations instead of difference models does not allow distinguishing metaelement updates from deletion/addition patterns. This problem is (partly) addressed in [6], which advocates for some metamodel difference management by means of *change traces*, although no specific proposal is adopted or given.

In [10] the authors discuss the possibility to induce model transformations through model weaving. In particular, weaving links are given to establish correspondences (or matchings) between metamodel elements and consequently to derive mappings between corresponding models. If the weaving is seen as a difference representation, the induced transformation can be considered as the automated co-adaptation of existing instances. Nonetheless, the approach in [10] lacks of expressiveness, since only additions and deletions can be represented through the semantics provided by the proposed weaving relationships. The problem of *metamodel matching* is also discussed in [5] where techniques based on schema matching algorithms are used to compute metamodel alignments.

The co-evolution problem is also investigated in the context of database evolution and metadata handling, which have been demonstrated to share several problems related to model management [11]. In fact, when schemas evolve to overcome new requirements all the interconnected artefacts need to be co-adapted, like queries, scripts and even existing data. Also in this field, a common solution relies on the separation between schema manipulations causing no or limited updates to existing instances versus modifications requiring deep structural changes and data conversions. Analogously to model co-evolution, simple situations can be automatically supported, while complex ones demand for user intervention, even though the environment can be adequately started-up [12].

2.2 Supporting Complex Metamodel Changes

A common aspect that seems to underlay current approaches to co-evolution is the atomicity of the changes, i.e. the classified change types are assumed to occur individually, which is not always the case since modifications tend to occur with arbitrary multiplicity and complexity. Additionally, interdependencies may also be present posing severe difficulties in distinguishing the various change types. To clarify such problems the sample evolution of the (simplified) Petri Net metamodel depicted in Figure 1 will be considered in the rest of the section. In particular, the initial metamodel $\mathtt{MM}_0$ consists of `Places` and `Transitions`, places can have source and/or destination transitions, whereas transitions must link source and destination places (`src` and `dst` association roles, respectively). In the new metamodel $\mathtt{MM}_1$, each `Net` has at least one `Place` and one `Transition`. Besides, arcs between places and transitions are made explicit by extracting `PTArc` and `TPArc` metaclasses, thus allowing to add further properties to relationships between places and transitions. Since `PTArc` and `TPArc` both represent arcs, they have been generalized in $\mathtt{MM}_2$ by the new abstract class `Arc` encompassing the integer metaproperty `weight`. Finally, the metaclass `Net` has been renamed into `PetriNet`.

The modifications applied to the Petri Net metamodel $\mathtt{MM}_0$ to obtain $\mathtt{MM}_1$ consists of breaking and resolvable changes. In fact the addition of the new `PTArc` and `TPArc` metaclasses breaks the conformance of the existing models to $\mathtt{MM}_0$ since, according to the new metamodel $\mathtt{MM}_1$, `Place` and `Transition` instances have to be related through `PTArc` and `TPArc` elements. However, models can be automatically migrated by adding for each couple of `Place` and `Transition` entities two additional `PTArc` and `TPArc` instances between them. An automatic model adaptation cannot be performed when $\mathtt{MM}_1$ is changed to get $\mathtt{MM}_2$ because of the breaking and unresolvable modifications. In particular, in this case, only a human intervention can introduce the missing information related to the weight of the arc being specified, or otherwise default values have to be considered.

All the scenarios of model co-adaptations, like the one of the Petri Net example, can be managed with respect to the possible metamodel modifications which can be

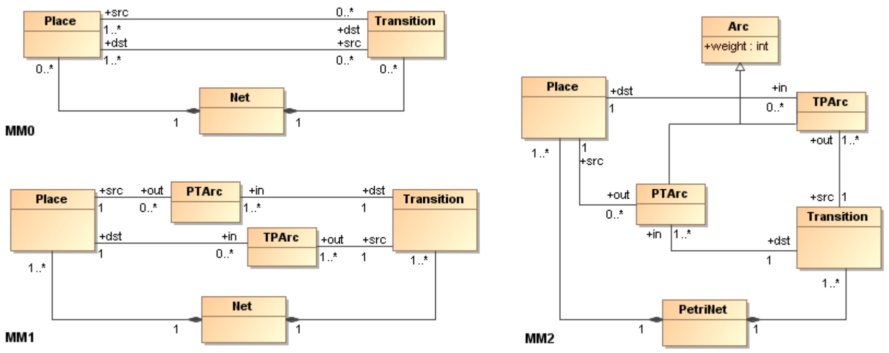


Fig. 1. Petri Net metamodel evolution

distinguished into *additive*, *subtractive*, and *updative* [8]. By going into more details, with additive changes we refer to the following metamodel element additions:

- *Add metaclass or metaproperty*, introducing new metaclasses or metaproperties is a common practice in metamodel evolution which gives place to metamodel extensions;
- *Generalize metaproperty*, a metaproperty is generalized when its multiplicity or type are relaxed, for instance the cardinality is modified from `3..n` to `0..n`, or a type is substituted with its supertype;
- *Pull metaproperty*, a metaproperty `p` is pulled into a superclass `A` and the old one is removed from a subclass `B`;
- *Extract superclass*, a superclass is extracted in a hierarchy and a set of properties is pulled on.

Subtractive changes consist of the deletion of some of the existing metamodel elements:

- *Eliminate metaclass*, a metaclass is deleted by giving place to a sub metamodel of the initial one;
- *Eliminate metaproperty*, a property is eliminated from a metaclass, it has the same effect of the previous modification;
- *Push metaproperty*, pushing a property in subclasses means that it is deleted from an initial superclass `A` and then cloned in all the subclasses `C` of `A`;
- *Flatten hierarchy*, to flatten a hierarchy means eliminating a superclass and introducing all its properties into the subclasses;
- *Restrict metaproperty*, a metaproperty is restricted when its multiplicity or type are enforced, for example the cardinality is modified from `0..*` to `0..10`, or a type is substituted with one of its subtypes.

Finally, a new version of the model can consist of some updates of already existing elements leading to updative modifications:

- *Change metaproperty type*, the type of a metaproperty is updated and the new type has not particular relationships with the old one;
- *Rename metaelement*, a metaelement is renamed;
- *Move metaproperty*, it consists of moving a property `p` from a metaclass `A` to a metaclass `B`;
- *Extract/inline metaclass*, extracting a metaclass means to create a new class and move the relevant fields from the old class into the new one. Vice versa, to inline a metaclass means to move all its features into another class and delete the former.

Such classification plays a key role in a transformational approach to model co-evolution presented by the authors in [8] and its discussion goes beyond the purpose of this paper; nonetheless, an overall illustration of such proposal is given in Figure 2. The implementation of the approach relies on the KM3 metamodeling language [13] which provides metamodeling constructs consisting of a common subset of OMG/MOF and EMF/Ecore. The applicability of the proposed co-evolution approach with respect to the metamodeling elements which are not included in such a subset is an open issue and it will be investigated in the near future. In particular, given two versions $\mathtt{MM}_1$ and $\mathtt{MM}_2$

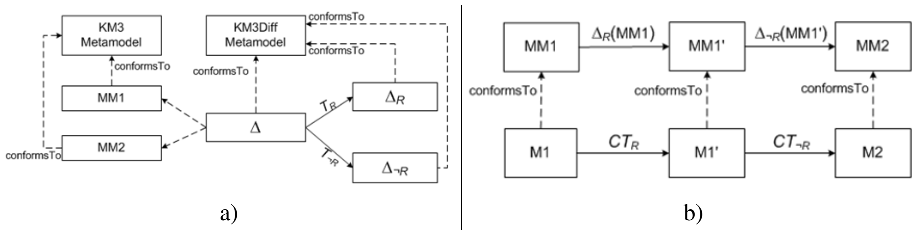


Fig. 2. Transformative co-evolution approach

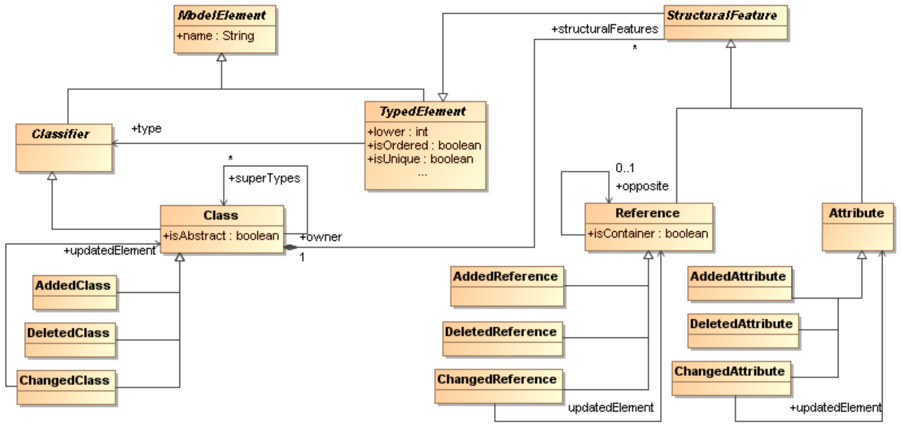


Fig. 3. Fragment of the generated difference KM3 metamodel

of the same metamodel, their differences are recorded in a difference model Δ, whose metamodel KM3Diff is automatically derived from KM3 and shown in Figure 3. Essentially, for each metaclass MC of the KM3 metamodel, the additional metaclasses AddedMC, DeletedMC, and ChangedMC are generated in order to represent additions, deletions, or changes, respectively, of MC instances [14].

In realistic cases, the metamodel modifications represented in the model Δ consist of an arbitrary combination of the atomic changes in Tab. 1. Hence, a difference model formalizes all kind of modifications, i.e. non-breaking, breaking resolvable and unresolvable ones. In this respect, the adopted difference representation approach is crucial. In particular, if the representation of the updates is too coarse-grained, then the co-adaptation acts with less efficacy. For instance, if the introduction of PTArc and TPArc in the sample $\texttt{MM}_0$ would be represented as the deletion of the current associations and the addition of those new entities (instead of an update of the current relationships), all the existing connections between arcs and transitions would be lost in the co-adaptation process. In fact, PTArc and TPArc would be interpreted as new relationships between arcs and transitions instead of being a refinement of them. In this respect, the quality of the approach used for the difference calculation may affects the results of the proposed co-adaptation technique. In other words, depending on

Table 1. Changes classification

Change type	**Change**
Non-breaking changes	Generalize metaproperty, Add (non-obligatory) metaclass, and Add (non-obligatory) metaproperty
Breaking and resolvable changes	Extract (abstract) superclass, Eliminate metaclass, Eliminate metaproperty, Push metaproperty, Flatten hierarchy, Rename metaelement, Move metaproperty, and Extract/inline metaclass
Breaking and unresolvable changes	Add obligatory metaclass, Add obligatory metaproperty, Pull metaproperty, Restrict metaproperty, Change metaproperty type, and Extract (non-abstract) superclass

the metamodels being considered, difference algorithms have to be properly chosen or customized. Interested readers can refer to [15] which summarizes the already existing approaches for model matching. Once the metamodel changes have been calculated (as for instance in [5]) and represented in Δ, such a difference model is automatically decomposed in two disjoint (sub) models, Δ_R and $\Delta_{\neg R}$ [8], which denote breaking resolvable and unresolvable changes, respectively. The decomposition is given by two model transformations, T_R and $T_{\neg R}$ (see Figure 2.a).

As previously said, the possibility to have a set of dependencies among the several parts of the evolution makes the updates not always distinguishable as single atomic steps of the metamodel revision. In such situations, a certain set of delta entities can pertain to multiple modification categories in Tab. 1 at the same time, and then the order in which such manipulations take place matters. In fact, it does not allow the decomposition of a difference model in Δ_R and $\Delta_{\neg R}$, like for instance when evolving $\mathtt{MM}_0$ directly to $\mathtt{MM}_2$ in Figure 1 (although the sub steps $\mathtt{MM}_0 - \mathtt{MM}_1$ and $\mathtt{MM}_1 - \mathtt{MM}_2$ are directly manageable). In these cases Δ_R and $\Delta_{\neg R}$ are said to be *parallel dependent* and they have to be further refined to identify and isolate the interdependencies causing the interferences. If Δ_R and $\Delta_{\neg R}$ are *parallel independent* then corresponding co-evolutions are generated separately. In particular, co-evolution actions are directly obtained as model transformations from the calculated metamodel changes by means of higher-order transformations, i.e. transformations which produce other transformations [16]. More specifically, two different higher-order transformations $\mathcal{H}_R$ and $\mathcal{H}_{\neg R}$ take Δ_R and $\Delta_{\neg R}$ and produce the (co-evolving) model transformations CT_R and $CT_{\neg R}$, respectively. Since Δ_R and $\Delta_{\neg R}$ are parallel independent CT_R and $CT_{\neg R}$ can be applied in any order because they operate to disjoint sets of model elements (see Figure 2.b). On the contrary, parallel dependence is more complex to manage: the main problem in having such kind of interdependencies is in the *nondeterminism* given by the following

$$\Delta_R | \Delta_{\neg R} \neq \Delta_R; \Delta_{\neg R} + \Delta_{\neg R}; \Delta_R$$

denoting with $+$ the nondeterministic choice. In the next section, we proposes a dependency analysis and resolution criteria to decompose and schedule the modifications in order to resolve the dependencies according to a comprehensive classification of them as they can occur in a metamodel evolution.

3 Dealing with Parallel Dependent Changes

The automatic co-adaptation approach recalled in the previous section relies on the parallel independence of breaking resolvable and unresolvable modifications. For instance, when evolving the sample PetriNet metamodel $\mathtt{MM}_0$ in Figure 1 directly to $\mathtt{MM}_2$, the approach cannot be directly applied unless the dependent changes in Δ_R and $\Delta_{\neg R}$ are identified and resolved. In particular, in the example, the *Add obligatory metaclass* modification, consisting of the addition of the attribute `weight` in the metclass *Arc*, depends on the addition of this new metaclass induced by the *Extract abstract metaclass* change. Such a dependence is due to the reference `owner` which, according to the KM3 metamodel, needs to be specified for each structural feature.

Being more precise, our solution is based on the following observation: given two versions of a same metamodel and a model Δ which represents their differences, the models Δ_R and $\Delta_{\neg R}$ obtained from the decomposition of Δ to isolate breaking resolvable and unresolvable modifications, respectively, are parallel dependent when the source and the target elements of the following references (defined in the KM3 difference metamodel) are not in the same difference model:

- $owner : StructuralFeature \rightarrow \{AddedClass, ChangedClass\}$, all the attributes and references defined in a given metamodel are related to a corresponding class which represents their owner. If a given structural feature *sf* belongs to Δ_R (or $\Delta_{\neg R}$) and its owner metaclass *mc* to $\Delta_{\neg R}$ (or Δ_R), then a parallel dependence occurs. In this case, *owner(sf)* can be specified once *mc* has been added or modified;
- $type : TypedElement \rightarrow \{AddedClass, ChangedClass\}$, given an element *te*, *type(te)* refers to the added or modified classifier *mc* which represents its type. In this respect, if a typed element *te* belongs to Δ_R (or $\Delta_{\neg R}$) and its type *mc* to $\Delta_{\neg R}$ (or Δ_R), then a parallel dependence occurs. In this case, *type(te)* can be specified once *mc* has been added or modified;
- $superTypes : Class \rightarrow \{AddedClass, ChangedClass\}^*$, in order to specify hierarchies of classes, the *superTypes* reference is available to define all the superclasses c_i of a given class c. If a given class c belongs to Δ_R (or $\Delta_{\neg R}$) and its superclasses c_i to $\Delta_{\neg R}$ (or Δ_R), then a parallel dependence occurs. In fact *superTypes(c)* can be specified once the superclasses c_i have been added or modified.

Because of such references, many of the metamodel changes recalled in the previous section may give place to parallel dependencies which are summarized in Table 2. In particular, the rows of the table reports unresolvable changes whereas the resolvable ones are given in the columns. Non empty cells represent the dependencies which may occur because of the corresponding couple of unresolvable and resolvable changes which might interfere one with another because of the specified reference. For instance, the cell `B1` is not empty since an *Add obligatory metaproperty* modification and an *Extract abstract superclass* one may give place to a dependence because of the references *owner* or *type*. In particular, an added obligatory metaproperty may have as owner or type the new superclass obtained by means of an *Extract abstract superclass* modification. In this respect, as in the PetriNet example, the dependence can be sorted out by applying the resolvable change before the unresolvable one (this is the meaning of the `R` in the cell `B1`).

Table 2. Metamodel change dependencies

		Resolvable changes			
		(1) Extract (abstract) superclass	(2) Push metaproperty	(3) Move metaproperty	(4) Extract/inline metaclass
Unresolvable changes	(A) Add obligatory metaclass	*R,¬R (superTypes)*	*¬R (owner)*	*¬R (owner)*	-
	(B) Add obligatory metaproperty	*R (owner,type)*	-	-	*R (owner,type)*
	(C) Pull metaproperty	*R (owner)*	-	-	-
	(D) Extract (non abstract) superclass	*R,¬R (superTypes)*	-	*¬R (owner)*	*R,¬R (superTypes)*
	(E) Change metaproperty type	*R (type)*	-	-	*R (type)*

The rest of the section is organized as follows: all the metamodel change dependencies summarized in Table 2 will be described in Section 3.1. The identification and the resolution of the dependencies occurring in large difference models are discussed in Section 3.2.

3.1 Classification of Change Dependencies

The description of the parallel dependent changes summarized in Table 2 exploits the sample metamodel evolution reported in Figure 4 (for the sake of readability, parallel independent combinations have not been included in that table). The differences between the sample metamodels MM_1 and MM_2 are represented in the difference model in Figure 5 which has been decomposed in the corresponding Δ_R and $\Delta_{\neg R}$ in Figure 6.

A1. Both the *Add obligatory metaclass* and *Extract abstract superclass* modifications give place to new metaclasses. Such modifications are parallel dependent if the metaclass added by the former is subclass of the metaclass added by the latter (or viceversa). For instance, in the running example, an *Add obligatory metaclass* modification has been executed to add the new metclass MC7 as specialization of MC4 which is a

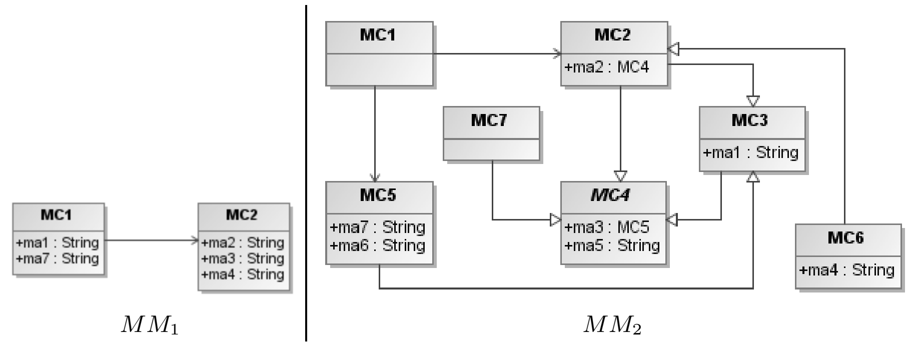


Fig. 4. Sample metamodel evolution

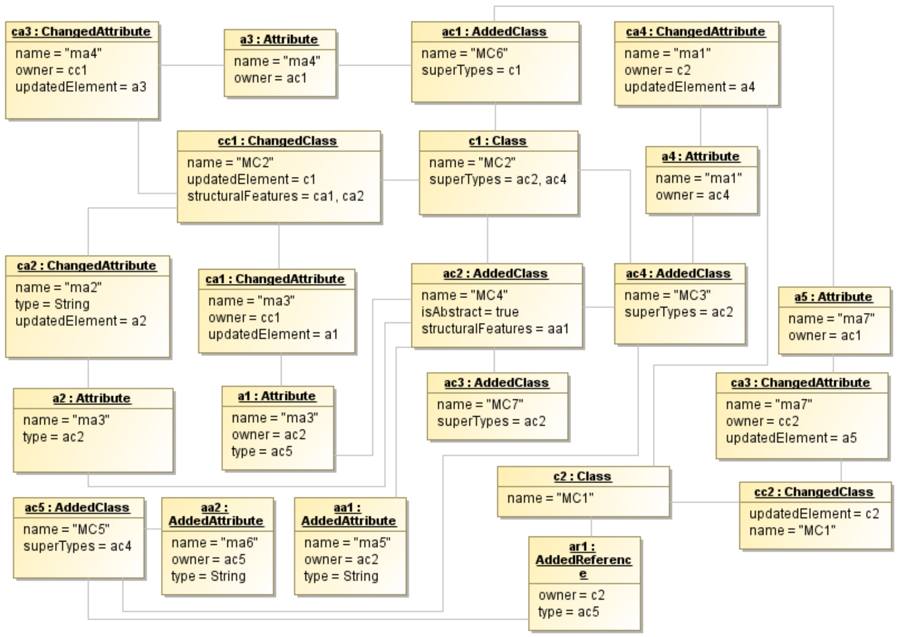
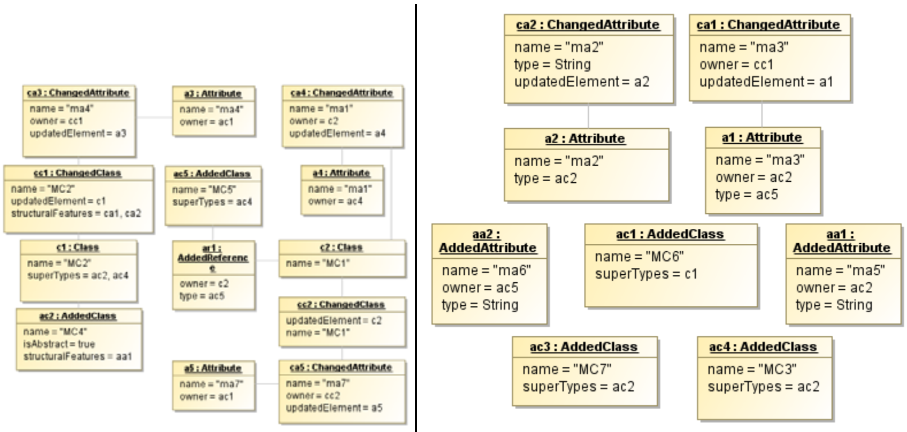


Fig. 5. Representation of the sample metamodel modifications



a. Resolvable changes (Δ_R) b. Unresolvable changes ($\Delta_{\neg R}$)

Fig. 6. Decomposed difference model

new abstract metaclass that has been added as superclass of the existing `MC2`. The addition of `MC7` is represented by the element `ac3` in the model $\Delta_{\neg R}$ whereas the addition of the metaclass `MC4` is represented in the Δ_R by means of the element `ac2`. Such modifications are parallel dependent since `supertTypes` of the added `MC7` refers to the metaclass `MC4` whose addition is in Δ_R.

B1. The owner or the type of a new attribute obtained by means of an *Add obligatory metaproperty* modification may be a new class which has been added by means of an *Extract abstract superclass* operation. For instance, in the running example the new meta attribute `ma5` has been added as represented by the element `aa1` in $\Delta_{\neg R}$ and its `owner` refers to the metaclass `MC4` which has been obtained through the *Extract abstract superclass* modification previously described.

C1. The *Pull metaproperty* modification moves a metaproperty `p` from a subclass `B` to the superclass `A`. If such superclass is obtained through an *Extract abstract superclass* modification, a parallel dependence occur since in order to set the reference `owner` of `p`, the metaclass `A` has to be added first. For instance, the metaproperty `ma3` has been moved from `MC2` to the new metaclass `MC4` by means of a *Pull metaproperty* modification (see the elements `ac2` and `a1` in $\Delta_{\neg R}$) Such modification depends on the addition of the metaclass `MC4` which is represented in Δ_R as described above.

D1. The *Extract non abstract superclass* modification extracts a non abstract superclass `A` in a hierarchy. If `A` is superclass of an abstract class obtained after an *Extract abstract superclass* modification (or viceversa), a parallel dependence is raised because of the `supertypes` reference. For instance, an *Extract non abstract superclass* modification has been performed to create the new metaclass `MC3` as superclass of `MC2` (see the element `ac4` in $\Delta_{\neg R}$). In this case, the dependence `D1` occurs since `MC3` also specializes the metaclass `MC4` (see the reference `superTypes` of the element `ac4` in $\Delta_{\neg R}$ to the element `ac2` in Δ_R) which has been obtained after an *Extract abstract superclass* modification represented in Δ_R.

E1. If the type of a metaproperty is changed to the abstract class obtained by means of an *Extract non abstract superclass* modification, a parallel dependence occurs because of the `type` reference. For instance, the type of the attribute `ma2` in the metaclass `MC2` has been changed from `String` to `MC4`. This is a *Change metaproperty type* modification and is represented in $\Delta_{\neg R}$ by means of the elements `ca2` and `a2`. However, since the new type of the attribute `ma2` is a class obtained by means of an *Extract abstract superclass* modification, the dependence *E1* occurs.

A2. The *Push metaproperty* modification deletes a metaproperty `p` from a superclass `A` and clones it in all the subclasses `C` of `A`. If the subclasses `C` have been added by means of *Add obligatory metaclass* modifications, parallel dependencies occur because of the `owner` reference. In the running example, an *Add obligatory metaclass* change has been performed to add `MC6` as specialization of `MC2`. Such a modification is represented in $\Delta_{\neg R}$ by means of the element `ac1`. Moreover, a *Push metaproperty* change has been executed to change the owner of the attribute `ma4` from the metaclass `MC2` to the just added `MC6`. This modification is represented in Δ_R by the elements `ca3` and `a3` instances of the metaclasses `ChangedAttribute`, and `Attribute`, respectively. The addition of `MC6` and the owner change of the attribute `ma4` are an example of the dependence *A2*.

A3. Similarly to the dependence *A3*, *A2* occurs because of the reference `owner` when a metaproperty is moved to a metaclass added by means of an *Add obligatory metaclass* modification. For instance, the attribute `ma7` has been moved from the metaclass `MC1` to the new metaclass `MC6` by means of the *Move metaproperty* change represented in Δ_R by the elements `ca5` and `a5`. Such a modification depends on the *Add obligatory metaclass* change which has to be performed first in order to add the metaclass `MC6` and update the value of the reference `owner` of the attribute `ma7`.

D3. A metaproperty can be moved to a new metaclass obtained by means of an *Extract non abstract superclass* modification. In this case, because of the `owner` reference, a dependence occurs and to set the owner of the moved property, the new non abstract metaclass has to be extracted first. In the running example, a *Move metaproperty* modification has been executed to move the attribute `ma1` from the metaclass `MC1` to `MC3` as represented by the elements `ca4` and `a4` in Δ_R. However, since the new owner of the attribute `ma1` is the metaclass `MC3` (obtained through an *Extract non abstract superclass* represented in $\Delta_{\neg R}$) the dependence *D3* takes place.

B4. The *Extract metaclass* operation means to create a new metaclass and move the relevant fields from the old metaclass to the new one and relate them. For instance, in Figure 4 an *Extract metaclass* operation has been performed to create the new metaclass `MC5` associated with the existing `MC1` (see the elements `cc2`, `c2`, `ar1`, and `ac5` in Δ_R). Consequently, if a new metaproperty `mp` is created by means of an *Add obligatory metaproperty* modification, a dependence with the *Extract metaclass* modification can be raised if the `type` or the `owner` of `mp` is the extracted metaclass. For instance, the new attribute `ma6` has been added in `MC5` as represented by the element `aa2` in $\Delta_{\neg R}$, and the modification depends on the *Extract metaclass* operation since the owner of the new attribute `ma6` is the extracted metaclass `MC5`.

D4. As previously said, the *Extract non abstract superclass* modification extracts a non abstract superclass `A` in a hierarchy. If `A` is superclass of a class obtained by means of an *Extract metaclass* modification (or viceversa), a parallel dependence is raised because of the `superTypes` reference. For instance, the metaclass `MC5`, obtained through an *Extract metaclass* modification, has been added as specialization of the class `MC3` which has been created by means of *Extract non abstract superclass* change giving place to dependent modifications.

E4. An existing metaproperty can be modified by setting its type to a metaclass which has been added by means of an *Extract metaclass* modification. In this case, dependent modifications have been performed which need to be sorted out. For instance, the type of the attribute `ma3` moved to the new metaclass `MC4` has been changed from `String` to the new metaclass `MC5` by means of a *Change mataproperty type* operation represented by the elements `ca1` and `a2` in $\Delta_{\neg R}$. Since the new type of the attribute is a class obtained by means the *Extract metaclass* modification, the dependence *E4* takes place.

When the evolution of a metamodel consists of complex modifications, the decomposition in resolvable and unresolvable changes can easily give place to dependencies which are usually difficult to be identified and sorted out by hand. In the next section we propose a formal approach to support the identification and the resolution of such dependencies.

3.2 Identification and Resolution of Change Dependencies

In this section we propose an approach to identify and resolve the dependencies which have been discussed in the previous section. The approach is based on the concepts of sets and functions which will enable a precise and formal identification and manipulation of dependencies among atomic changes.

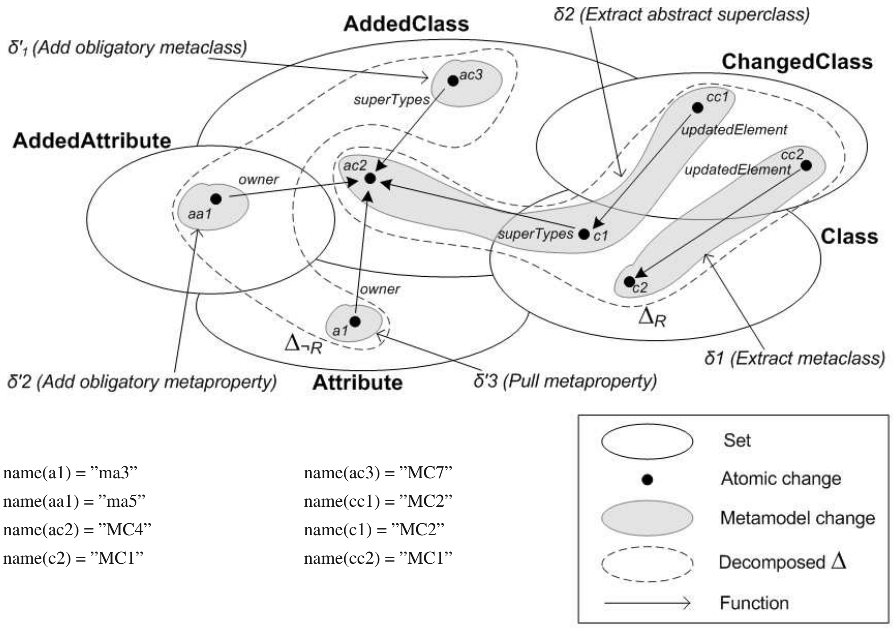
In particular, an algebra signature is directly derived from the KM3 difference metamodel whose elements define sorts and functions as reported in Figure 7. This operation can be performed in an automated way by means of model transformations as shown in [17,18]. More precisely, the metamodel induces the signature Σ composed of sorts (S) and functions (OP): for each non abstract metaclass of the metamodel a correspondent set in S is defined, and the functions in OP are induced by the attributes and references of all the metaclasses. For instance, the attribute `name` of the metaclass `Class` induces the definition of the function *name: Class → String*. Moreover, to specify the `type` of an `Attribute`, the function *type: Attribute → Class* is defined with respect to the property *type* of the abstract metaclass `TypedElement` which is superclass of the `Attribute` one.

The sets and the functions in Figure 7 enables the encoding of models conforming to the KM3 difference metamodel as in the example in Figure 8 which depicts the encoding of a fragment of the difference models in Figure 6. More specifically, the elements `ac3`, `aa1` and `a1` of Figure 6.a, `cc1`, `c1`, `ac2`, `cc2`, and `c2` of Figure 6.b are represented. The ovals in Figure 8 represents some metaclasses of the KM3 difference metamodel. The elements contained in such ovals are instances of the represented metaclasses. For example, the changed class `MC2` on the left hand side of Figure 6.a, is encoded in Figure 8 by means of the element `cc1` contained in the `ChangedClass` oval. Please note that the overlaps of the ovals and the graphical order in which they appear have no semantics and their layout is related to presentation purposes only.

The resolvable and the unresolvable modifications are also distinguished (see the dashed parts which enclose Δ_R and $\Delta_{\neg_R}$, respectively) and each of them consists of a set of the atomic metamodel changes described in the previous section. For instance, the modification δ_2 in Δ_R corresponds to the *Extract abstract superclass* modification which has been applied to the metamodel MM_1 in Figure 4 to add the metaclass `MC4` in

$$
\begin{array}{l}
\Sigma = (S, OP) \\
S := \{Class, AddedClass, ChangedClass, Attribute, \\
\quad AddedAttribute, ChangedAttribute, \dots\} \\
OP := \{\ name : Class \rightarrow String \\
\quad name : Attribute \rightarrow String \\
\quad isAbstract : Class \rightarrow Boolean \\
\quad isPrimary : Attribute \rightarrow Bool \\
\quad type : Attribute \rightarrow Class \\
\quad owner : Attribute \rightarrow Class, \dots\}
\end{array}
$$

Fig. 7. Fragment of the signature induced by the KM3 difference metamodel



name(a1) = ”ma3”	name(ac3) = ”MC7”
name(aa1) = ”ma5”	name(cc1) = ”MC2”
name(ac2) = ”MC4”	name(c1) = ”MC2”
name(c2) = ”MC1”	name(cc2) = ”MC1”

Fig. 8. Sample difference model encoding

MM_2. Moreover, the *Add obligatory metaclass* modification which has been executed to add the metaclass `MC7` has been represented by δ'_1 in $\Delta_{\neg R}$. As discussed in the previous section, the latter modification depends on the former according to the case `A1` in Table 2. Such a dependence can be noticed also by considering the encoding in Figure 8. In fact, the reference `superTypes` of the elements `ac3` in $\Delta_{\neg R}$ has `ac2` as value which is in Δ_R. In this respect, the modification δ'_1 depends on δ_2, hence $\Delta_{\neg R}$ depends on Δ_R.

Being more formal, by considering the *owner*, *superTypes*, and *type* functions defined at the beginning of the section, the following definitions can be given:

Definition 1. *Let* $\delta_1 = \{a_1, a_2, \ldots, a_n\}$ *and* $\delta_2 = \{b_1, b_2, \ldots, b_m\}$ *be two metamodel changes.* δ_1 depends on δ_2 *if there exists a couple* (a_i, b_j), $i \in \{1 \ldots n\}$, $j \in \{1 \ldots m\}$, *of atomic modifications such that* $owner(a_i) = b_j$ *or* $type(a_i) = b_j$ *or* $superTypes(a_i) = b_j$.

Definition 2. *Let* $\Delta_1 = \{\delta_1, \delta_2, \ldots, \delta_n\}$ *and* $\Delta_2 = \{\delta'_1, \delta'_2, \ldots, \delta'_m\}$ *be two difference models,* Δ_1 depends on Δ_2 *if there exists a couple* (δ_i, δ'_j), $i \in \{1 \ldots n\}$, $j \in \{1 \ldots m\}$, *of metamodel changes such that* δ_i depends on δ'_j.

It is important to stress how the functions above are part of the KM3 definition and are the only responsible for the dependencies among the breaking resolvable and breaking unresolvable changes. As a consequence, this makes the technique independent from the metamodel and its underlying semantics.

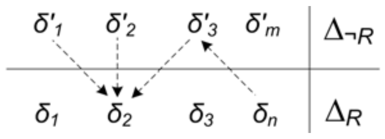


Fig. 9. Fragment of the sample change dependencies

As mentioned above, the automatic co-adaptation of models relies on the parallel independence of breaking resolvable and unresolvable modifications, or more formally

$$\Delta_R|\Delta_{\neg R} = \Delta_R;\Delta_{\neg R} + \Delta_{\neg R};\Delta_R \tag{1}$$

where $+$ denotes the non-deterministic choice. In essence, their application is not affected by the adopted order since they do not present any interdependencies. If change dependencies are identified they have to be sorted out in order to recover the parallel independence condition. In this respect, according to Table 2, the discovered dependencies induce the order in which changes have to be applied. For instance, Figure 9 contains a fragment of the sample metamodel changes presented above with their dependencies depicted by means of dashed arrows. By taking into account such dependencies and the resolution criteria presented above, the correct scheduling of modifications is as follows

$$(\Delta_R - \{\delta_n\}) \mid (\Delta_{\neg R} - \{\delta'_1, \delta'_2, \delta'_3\}) \ ; \ \{\delta'_1 \mid \delta'_2 \mid \delta'_3\} \ ; \ \delta_n \tag{2}$$

denoting with $-$ the calculation of model differences and with ; and | the sequential and parallel application of differences, respectively.

The identification of change dependencies can be easily automatized by translating each non-empty entry in Table 2 into first-order logic predicates. For instance, the dependency *B1* in Table 2 can be detected if *exists* a structural feature *sf* in the set `AddedAttribute` or `AddedReference` such that *owner(sf)* or *type(sf)* is an element belonging to the set `AddedClass` and which is an abstract superclass of one of the existing elements in the set `Class`. Thus the dependency identification can be implemented in OCL [19], for instance, which has the support for specifying first-order logic predicates.

Finally, it is worth to mention that cyclic change dependencies cannot occur. In particular, because of the typing of the functions *type*, *owner*, and *superTypes* the only admitted cycle might be caused by the last one since it has the set *Class* as domain and codomain. However, having a cyclic dependence because of such a function would give the possibility to define cyclic hierarchies which are not admitted in general.

4 Conclusions and Future Work

In this paper, we have presented an approach that automates the adaptation of models whenever the corresponding metamodel is subject to evolution, i.e., to arbitrary, complex and, possibly *non-monotonic* modifications. To the best of our knowledge, the existing approaches are only dealing with atomic changes which are assumed to occur in isolation and which can then be automatized in a pretty straightforward way. Complex modifications, which can be applied with arbitrary multiplicity and complexity, poses severe difficulties since they may present interdependencies which compromises the automation of the adaptation.

This work advocates the adoption of the transformational approach presented in [8] which encompasses the decomposition of difference models to distinguish among breaking resolvable and unresolvable metamodel changes. The main contribution of this paper is in providing a classification of the interdependencies which can occur in these two categories of modifications. The classification is used to define resolution criteria which provide the decomposition and the correct scheduling of modifications. Moreover, it has been shown how the dependencies are caused by features which are defined in the meta-metamodel (in this case KM3), which implies that the results are general and agnostic from the metamodel and its semantics.

A prototypical implementation of the co-evolution approach is available at [20]. Future works includes a more systematic validation of the dependency detection and resolution technique which necessarily encompasses larger population of models and metamodels. Finally, we plan to investigate how the works related to change impact analysis [21] can be adapted and used in MDE to support the co-evolution of metamodels and corresponding models.

References

1. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (2006)
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (2004)
3. Favre, J.M.: Meta-Model and Model Co-evolution within the 3D Software Space. In: Procs. of the Int. Workshop ELISA at ICSM (September 2003)
4. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
5. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 326–340. Springer, Heidelberg (2008)
6. Gruschko, B., Kolovos, D., Paige, R.: Towards Synchronizing Models with Evolving Metamodels. In: Procs of the Work. MODSE (2007)
7. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: Automatability of Coupled Evolution of Metamodels and Models in Practice. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 645–659. Springer, Heidelberg (2008)
8. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 12th IEEE International EDOC Conference (EDOC 2008), Munich, Germany, pp. 222–231. IEEE Computer Society, Los Alamitos (2008)
9. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. Journal of Visual Languages & Computing 15(3-4), 291–307 (2004)
10. Del Fabro, M.D., Valduriez, P.: Semi-automatic Model Integration using Matching Transformations and Weaving Models. In: The 22th ACM SAC - MT Track, pp. 963–970. ACM, New York (2007)

11. Bernstein, P.: Applying Model Management to Classical Meta Data Problems. In: Procs. of the 1st Conf. on Innovative Data Systems Research, CIDR (2003)
12. Galante, R., Edelweiss, N., dos Santos, C.: Change Management for a Temporal Versioned Object-Oriented Database. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 1–12. Springer, Heidelberg (2002)
13. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
14. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology 6(9), 165–185 (2007)
15. Kolovos, D.S., Di Ruscio, D., Paige, R.F., Pierantonio, A.: Different models for model matching: An analysis of approaches to support model differencing. In: Proc. 2nd CVSM 2009, ICSE 2009 Workshop, Vancouver, Canada (2009) (to appear)
16. Bézivin, J.: On the Unification Power of Models. Jour. on Software and Systems Modeling (SoSyM) 4(2), 171–188 (2005)
17. Di Ruscio, D.: Specification of Model Transformation and Weaving in Model Driven Engineering. Ph.D thesis, Università degli Studi dell'Aquila (February 2007), `http://www.di.univaq.it/diruscio/phdThesis.php`
18. Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report n. 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA) (April 2006)
19. Object Management Group (OMG): OCL 2.0 Specification, OMG Document formal/2006-05-01 (2006)
20. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Implementation of an automated co-evolution of models through atl higher-order transformations (2008), `http://www.di.univaq.it/diruscio/CoevImpl.php`
21. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)

Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages

Iván García-Magariño, Jorge J. Gómez-Sanz, and Rubén Fuentes-Fernández

Software Engineering and Artificial Intelligence
Facultad de Informática
Universidad Complutense de Madrid, Spain
ivan_gmg@fdi.ucm.es, jjgomez@sip.ucm.es, ruben@fdi.ucm.es
http://grasia.fdi.ucm.es/

Abstract. Practitioners of model-driven engineering usually find that producing model transformations still requires much effort. The Model Transformation By-Example (MTBE) approach softens this problem by means of the automated creation of model transformations from pairs of source and target model examples. However, the current techniques and tools for MTBE are limited in the sense that they can only produce transformations rules that are limited to just one element in some of the patterns. In addition, some transformation languages cannot directly represent many-to-many transformation rules. To overcome both limitations, this paper presents a MTBE algorithm, that can generate many-to-many rules in several transformation languages even if the transformation language cannot directly represent these kinds of rules. In particular, the implementation of this algorithm with the ATL language is available for practitioners, and it has already generated several MTs that are applied in software development. Finally, the paper compares this algorithm with existing techniques for MTBE.

Keywords: Model-driven engineering, model-driven development, model transformations, model-transformation by-example, algorithm.

1 Introduction

The Model-Driven Development (MDD) [11] approach moves the focus of the development from the code to the models of the system. It organizes development around the definition and transformation of models to generate all the required artifacts. For the specification of these Model Transformations (MTs), it has defined multiple Model Transformation Languages (MTLs). Some relevant examples are the *ATLAS Transformation Language* (ATL) [7], QVT [10], Tefkat [9], and VIATRA2 [14]. All these languages demand a large amount of effort from designers to define transformations. Their rules have to be coded as in traditional programming languages, which reduces the level of abstraction in the development when compared with modeling. Coding these rules implies studying the expected inputs and outputs, producing candidate transformation rules, and debugging them until they work as expected. Besides, designers have not much support for this development since the related tools are not as mature as the ones for widespread programming languages. For instance, transformation tools usually lack of editing assistants or debugging capabilities. Moreover, some designers are not used to the declarative approach followed by most MTLs.

A solution for this problem consists of using *Model Transformations By Example* (MTBE) [13]. In this approach, rules are automatically generated from fragments of models. Designers have to provide models that are examples of the inputs and outputs that the rule has to consider. The history of this technique dates back to the Krishnamurthi's work [8] about a *transformation by-example* proposal for XML, concretely the *XT3D* transformation system. Though this work does not mention model transformations, it can be considered the precursor of MTBE. Afterwards, Varro [13] introduced the MTBE term, and Varro and Balogh [15] presented a implementation of MTBE using inductive logic programming. These works initiated a line of research with followers like Wimmer et al. [17], who presented a MTBE generator for ATL, and Siikarla et al. [12], who proposed a process for the iterative definition of transformations through partial examples. The main limitation of these approaches is that they need the input pattern (sometimes also the output) to contain only one modeling element. That is, their rules cannot take data from several modeling elements. For instance, many-to-many rules that create attribute values by means of composition of attribute values from several modeling elements are necessary in agent-oriented software engineering, as one can observe in the evaluation presented in Section 4.

This paper contributes to MTBE with an algorithm that addresses this limitation. It deals with networks of concepts and not only one-to-one or one-to-many transformation rules between connected graphs. The generation of many-to-many rules is based on the creation of constraints for the simulation of input patterns of several elements. Also, this algorithm works with mappings embedded in the models. This means that models themselves indicate how information should be transferred from the input to the output.

This research has also developed a tool that implements this algorithm for the *ATLAS Transformation Language* (ATL). Due to the ATL requirements, this tool uses ECore [2] as language for defining the involved metamodels and models, and the *Object Constraint Language* (OCL) [16] for the constraints. The main reasons for the choice of ATL are: its the wide technical support [1] for Eclipse; and its wide use [3] due to its hybrid approach between declarative and imperative, and its mechanisms for modularization and traceability. Nevertheless, it must be noticed that this specific implementation is a proof of concept, since the algorithm could be adapted for other transformation languages that are hybrid between declarative and imperative approaches and use constraints in the input side of the rules, such as QVT, XSLT, Tefkat or VIATRA2. The tool can be downloaded from our web [6].

The remainder of the paper is organized as follows. Section 2 introduces some preliminary definitions used in the paper. Section 3 presents the proposed algorithm for MTBE, which overcomes some limitations of existing MTBE algorithms. Section 4 evaluates the algorithm by presenting both an implementation for ATL, the *MTGenerator* tool, and several MTs generated by the implementation. Section 5 compares the presented algorithm with other existing relevant approaches for MTBE. Finally, the paper discusses some conclusions and future work in Section 6.

2 Background Definitions

This section introduces some basic definitions used in the remainder of the paper. They refer to the notions of metamodel, model and element considered in the algorithm for MTBE.

Definition 1. *A **model*** M *is a tuple* $M =< E, G, R, A, O, O_R >$ *that is subject to the following conditions:*

- $\mathbf{E}$*, called **elements**, is a set of tuples* $\{< n, t > |n \in \mathcal{I}, t \in \mathcal{T}\}$*, where* $\mathcal{I}$ *is the set of valid identifiers and* $\mathcal{T}$ *the set of valid types.*
- $\mathbf{G}$ *is a set of tuples* $\{< e_1, e_2, n, t > |e_1, e_2 \in \mathbf{E}, t \in \mathcal{T}, n \in \mathcal{I}\}$ *called **aggregation relations**. In this set, there is no sequence* $e_1 e_2 \ldots e_{n-1} e_n$ *such that* $e_1 = e_n$ *and* $< e_i, e_{i+1}, n, t > \in \mathbf{G}$*, i.e., there are no cycles formed by aggregation relationships.*
- $\mathbf{O_R} \in \mathbf{E}$ *is the* model root *forced by the realization of the model in different implementations. Reviewed MTLs process models expressed in XML, which forces always to have a single root. This root appears as a distinguished entity in the model. The model root is optional and regards to the selected implementation language. For instance, QVT uses* graph-based *models, which have not a forced model root.*
- $\mathbf{O}$ *is a set of elements* $e \in \mathcal{E}$ *such that* $(\exists < O_R, e, n, t > \in O_R.G,\ if\ \exists O_R)$ *or* $(\nexists < O_R, e, n, t > \in G,\ if\ \nexists O_R)$*. These elements are referred as* logical roots.
- $\mathbf{R}$ *is a set of tuples* $\{< e_1, e_2, n, t > |e_1, e_2 \in \mathbf{E}, t \in \mathcal{T}, n \in \mathcal{I}\}$ *called **references**.*
- $\boldsymbol{A}$ *is a set of pairs* $\{< e, v, n, t > |e \in \mathbf{E}, v \in \mathcal{T_P}, n \in \mathcal{I}\}$ *named **attributes**, where* $e \in E$*,* v *is a value of primitive type,* n *denotes its name, and* t *denotes its type. The set* $\mathcal{T_P}$ *contains all primitive types like integer, string, boolean, and so on.*

Definition 2. *A **metamodel** declares how a model is constructed. Given a tuple* $M =< E, G, R, A, O, O_R >$*, a metamodel would declare the set of valid types that can be used in* $\mathbf{E}, \mathbf{G}$*, and* $\mathbf{R}$*. It declares as well the set of valid type aggregation or reference relationships,* $\mathbf{G}$ *or* $\mathbf{R}$*, among the set of valid entities types,* $\mathbf{E}$*. Finally, it declares the attributes of each entity in* $\mathbf{E}$*. A model is produced from a metamodel by creating entities of the types declared by the metamodel and connecting them with relationships allowed by the metamodel.*

Given this definitions of metamodel and model, this work considers the following notations to work with them.

Notation 1. *Given a* ***model*** $M :=< E, G, R, A, O, O_R >$, *M.E, M.G, M.R, M.A, M.O and* $M.O_R$ *respectively denote E, G, R, A, O and O in model M.*

Notation 2. *Given a model* $M :=< E, G, R, A, O, O_R >$, *an* ***element*** $e \in E$, *an* ***aggregation relation*** $g \in G$, *and a* ***reference*** $r \in R$,

- *e.G denotes* $\{< e_1, e_2, n, t >\in G : e = e_1\}$
- *e.R denotes* $\{< e_1, e_2, n, t >\in R : e = e_1\}$
- *e.A denotes* $\{< e_x, v_x >\in A : e = e_x\}$
- $e.g.e_2$ *denotes* $e_2 : g =< e, e_2, n, t >\in G$
- $e.r.e_2$ *denotes* $e_2 : r =< e, e_2, n, t >\in R$

The components of models are usually labeled with names. According to this reality, the following notation is used.

Notation 3. *Given a model* $M :=< E, G, R, A, O, O_R >$, *an* ***element*** $e \in E$, *and an* ***aggregation relation*** $g \in G$,

- *e.n denotes* $(n : e =< n, t >)$, *which is the name of the element*
- *g.n denotes* $(n : g =< e_1, e_2, n, t >)$, *which is the name of the aggregation relation*

Each MTL has its own notation; consequently, this work uses a generic grammar capturing the most relevant from several transformation languages. The grammar is described using EBNF; bold words represent terminal symbols, whereas non-bold words represent non-terminal symbols:

1: MT::= **header** DefVariables **begin** Rules **end**
2: DefVariables::= DefVariable DefVariables
   DefVariables::= DefVariable
   DefVariable::=**Define** Name : Type
3: Type ::= List
4: Rules ::= Rule Rules
   Rules ::= Rule
5: Rule::= **rule** InputElement (Constraint) **to** OutputElements
   imperative Actions **endrule**
6: Constraint::= Constraint **and** Constraint
   Constraint::= Constraint **or** Constraint
   Constraint::= **exist**(Element)
   Constraint::= **attributeConstraint**(Attribute)
   Constraint::= **aggregationConstraint**(Aggregation)
   Constraint::= **referenceConstraint**(Reference)
   Constraint::= **BasicRootConstraint**
7: OutputElements::= Element (Actions) , OutputElements
   OutputElements::= Element (Actions)

```
 8: Expression::= Element→Attribute→Expression
    Expression::= Element→Aggregation→Expression
    Expression::= Element→Reference→Expression
    Expression::= Element
    Expression::= Value
 9: Actions ::= Actions Action
    Actions ::= Action
10: Action ::= assign(Aggregation, Variable)
    Action ::= assign(Attribute, Variable)
```

The *Element* in line 8 will be an element of the model. *Attribute*, *Variable*, and *List* non-terminal symbols are not defined here, but they refer to their intuitive meaning. The genericity of this grammar is shown in Table 1, which studies several transformation languages and the existence of the primitives remarked in the grammar. As one can observe, required constraints are available in all of them, and only the multiple input possibility is missing in some cases.

Table 1. Properties Transformation Languages

	Constraints				Actions	Rule patterns	
	exist	attribute	aggregation	reference	assign	multiple input	multiple output
QVT	√	√	√	√	√	X	√
ATL	√	√	√	√	√	X	√
XSLT	√	√	√	√	√	X	√
Tefkat	√	√	√	√	√	√	√
VIATRA2	√	√	√	√	√	√	√

Finally, the description of the algorithm uses also a pseudocode language, whose particularities are:

– The **replace(str,s,t)** function returns a string with the same value as *str* but where the target *t* replaces the appearances of the source *s*.
– The notation <**non-terminal**> indicates a string type derivable from the non-terminal of the aforementioned grammar.

3 The MTBE Algorithm

The presented algorithm produces a MT that contains rules transforming a group of elements into another group of elements. The algorithm can be implemented in most of the transformation languages since it is based on the primitives present in all of them (see Table 1). Though some languages support transformation rules with multiple input patterns, these languages happen to be of reduced use in the community. Hence, the utility of this algorithm for those languages is straightforward. This missing functionality is provided by means of constraints which are common to all transformation languages. To ensure enough abstraction and genericity, the algorithm sticks to the definition of MT presented in the previous section. The composition of the different words of the MT is made by means of the $\oplus$ operator. Terminal nodes of the MT grammar appear in the algorithm as bold text. The algorithm is introduced below in a high level of abstraction.

function mtbe(mm_i, mm_o : **metamodel**; pairs: $\mathcal{P}^{M_{mm_i} \times M_{mm_o}}$):MT

The algorithm takes as input two metamodels, mm_i and mm_o, and pairs of instances of those metamodels. These pairs are a subset of $\mathcal{P}^{M_{mm_i} \times M_{mm_o}}$, which stands for the set of sets whose elements are pairs of models instantiating the input models, i.e., $\{< m_i, m_o > | m_i$ is an instance of mm_i, m_o is an instance of $mm_o\}$. The output of the algorithm is a MT satisfying the grammar introduced in the previous section.

```
1: begin
2: dictionary = ∅
3: vars = GenerateVariables(mm_o)
4: rules = ∅
5: for each < m_i, m_o >∈ pairs do
6:     main := SelectMainElement(m_i.E)
7:     UpdateDictionary(dictionary, main, BasicRootConstraint)
8:     inCons := GenerateInputConstraints(main)
9:     ... Continues later... ;
```

The information transfer from the input models to the output models is achieved by means of a dictionary. A **dictionary** is a set of tuples $< e_x, c_x >$, where $e_x \in E$ is an element and c_x a constraint expression of the MTL.

The algorithm starts with an empty dictionary, a list of variables, and a preliminary root element of the model. If there are several possible roots, then anyone is chosen. This root element is traversed until reaching all connected elements by means of references, aggregation, or attribute relationships.This information is added as constraints to the *inCons* variable, which represents the input constraints of the current rule.

```
10: ...
11: for each < m_i, m_o >∈ pairs do
        ... Continuing main loop... ;
12:     for each potentialMain ∈ m_i.E such that potentialMain is a root
13:         potentialMainConstraint = CreateConstraint(potentialMain)
14:         UpdateDictionary(dictionary, potentialMain, potentialMainConstraint)
15:         inCons = inCons and GenerateInputConstraints(potentialMain)
16:     end for
17:     ... Continues later ... ;
```

After a reference point is chosen, *main* in this case, other possible roots are studied and included as constraints in the rule. These new elements are added to the dictionary and to the input constraints. So far, all input elements have been taken into account in *inCons*, and it is the turn of considering output elements. The variable *inCons* is constructed following the grammar rule for MTLs in Line 6.

```
18: ...
19: for each $< m_i, m_o >\in pairs$ do
        ...Continuing main loop...;
20:     $aggregatedToRootElements = \varnothing$
21:     $outElems = \varnothing$
22:     for each $e \in m_o.O.e$ do
23:         $outElems := outElems \oplus$ GenerateOutputElement($dictionary$, e)
            $aggregatedToRootElements := aggregatedToRootElements \cup \{e\}$
24:     end for
25:     $actions =$ GenerateActions ($aggregatedToRootElements$)
26:     $rule =$ rule main( $\oplus$ $inCons$ $\oplus$ ) $\oplus$ to $\oplus$ $outElems$ $\oplus$
27:             imperative $\oplus$ $actions$ $\oplus$ endrule
28:     $rules = rules \oplus rule$
29: end for
```

This part of the algorithm visits all elements which are roots in the output model, i.e., the elements of $m_o.e$. For each one of these elements, a set of MTL elements, following the MTL *actions* rule in Line 7, is generated. There is a set because, from the selected root, all connected elements are visited. This set contains imperative actions intended to fill in values in the attributes of generated elements. Afterwards, a basic rule is composed using the MTL rules in Lines 4 and 5 from previous section. The emphasized text represents pieces of information stored in variables. Rules are chained one to the other with $\oplus$ operator to form the final MT.

```
30: ...
31: rootRule = GenerateRootRule()
32: rules = rules $\oplus$ rootRule
33: mt = header $\oplus$ vars $\oplus$
34:         begin $\oplus$ rules$\oplus$ end
35: return mt
36: end
```

The final transformation is constructed using the variables initially extracted and the rules obtained for each pair of models. The combination of rules to form the actual transformation follows the MTL grammar rule in Line 1.

3.1 Allocation of Target Elements

The simulation of many-to-many transformation rules implies that the number of elements may be different from the source model to the target model. In most

MTLs, the allocation of the target elements must be explicitly managed when using these kind of transformation rules.

This algorithm manages the allocation of target elements by defining a helper variable for each element of the target model aggregated from its root (see Line 3 in the algorithm and Line 37 for the related function). Then, the generated rules add the original target elements to these variables. The function *GenerateActions* (see Line 43) takes as input these variables and "inserts" (see Line 46) the target elements (i.e. *entry.g.name*) in them (i.e. *entry.id*). Finally, this algorithm creates a last rule (see Line 49) that transforms the source model root into the target model root, and it adds all the target elements in the target model root through the correspondent actions in the imperative section of the rule.

```
37: function GenerateVariables(MMB: metamodel): $DefVariables$
38: vars := $\varnothing$
39: for each t $\in$ MMB.E do
40:     vars := vars $\oplus$ {t $\oplus$ : List}
41: end for
42: return vars
```

The function builds a list of variables named exactly as the types mentioned in the metamodel for valid entities.

```
43: function GenerateActions($elements$:E): $Actions$
44:         actions = $\varnothing$
45:         for each entry $\in$ elements do
46:                 actions = actions $\oplus$ insert( $\oplus$ entry.g.name $\oplus$ entry.id $\oplus$ )
47:         end for
48: return actions
```

For each element, an insert action is produced.

```
49: function GenerateRootRule (ma is instance of $mm_i$, mb is instance of $mm_o$): $Rule$
50: rule:= rule begin $\oplus$ $ma.O_R$ $\oplus$ () to $mb.O_R$ $\oplus$ imperative
51: for each g $\in$ $mb.O_R.G$ do
52:     rule := rule $\oplus$ assign($mb.O_R$ , g.t )
54: end for
55: rule:= rule $\oplus$ endrule
56: return rule
```

3.2 Simulation of the Input Side of Rules by Means of Constraints

Some existing MTLs cannot directly define rules that receive input from several non-connected graphs of elements. For this reason, this algorithm simulates the input side of rules by means of constraints. One of the elements of the input model is selected as the *main* element, which is the single input element of the corresponding rule. Other input elements are included in the rule through constraints that depends on the *main* element.

The algorithm for generating these constraints is recursive. The function concatenates the constraint resulting from the recursive call with the reference expression that contains the main element. In this manner, the constraint expressions with the *Transformation IDentifiers* (TIDs) can be added to the dictionary as pairs, and then used in the propagation of attributes from source models to target models. The TIDs are attributes of the input elements that are uniquely associated with input elements, and they are used for the matching mechanism, which is described in Section 3.4.

```
57: function GenerateInputConstraints (e: element): Constraint
58: inCons:=exist( e )
59: for each a ∈ e.A do
60:         inCons := inCons ⊕ and attributeConstraint( ⊕ a ⊕)
61: end for
62: for each r ∈ e.R do
63:         inCons := inCons ⊕ and referenceConstraint( ⊕ r ⊕ )
64:         GenerateInputConstraints($e.r.e_2$)
65: end for
66: for each g ∈ e.G do
67:         inCons := inCons ⊕ and aggregationConstraint( ⊕ g ⊕ )
68:         GenerateInputConstraints( $e.g.e_2$)
69: end for
70: return inCons
```

The GenerateInputConstraints reproduces the structure of the input model by means of *attributeConstraints, referenceConstraint*, and *aggregationConstraint*. Recursively, the algorithm traverses all elements interconnected with reference or aggregation relationships. Infinite recursive loops are avoided by the properties requested from a model, i.e., there are no loops created by references or aggregation relationships.

The generation of constraints in the algorithm using the function *GenerateInputConstraints* is achieved by traversing each main element of the input example model (see Lines 12-16) as follows:

1. Starting from the main element, a new constraint is created to check the value of each attribute. In the input model, attributes can have string or integer literals as values. The algorithm interprets a specific value, as the need of having that exact value in the input model. If the attribute has empty values, which can be the empty string and zero or MAXINT[1] regarding the user preferences for integers for instance, it is assumed that the attribute can have any value; in other words, it is a *wildcard*. Wildcards are translated in the corresponding constraint as 'true' literals.

[1] Maximum value of the integer type.

2. The function creates another constraint for each reference (or aggregation) coming out from the main element. This new constraint intends to capture the dependencies among input elements, so that the generated rule can match similar structures. This constraint requires knowing the path from the main element to the current element. A path here would be the set of concrete references traversed in order to go from the main element to the current element. The cardinality of a reference can be one or more than one. Regarding the selected MTL, the distinction between these two cases with additional constraints may be necessary. The traversal is recursive, and cycles can be avoided in different manners regarding the implementation.

This process allows considering different kinds of configurations in the input model example. In the simplest case, all the elements in the input model can be reached from one main element. This is the common setting that existing MTBE proposals consider. However, there are worse cases where there can be non-connected graphs of concepts. This traversal gets more complicated since there are several main elements, which are not necessarily linked. In that case, the procedure previously mentioned is applied for each identified potential main element (see Lines 12-16) .

3.3 Generation of the Output Side of the Rule

The output side of the rule in most MTLs (see Table 1) can have several modeling elements. Hence, there are no obvious limitations to the complexity of the output model but explicitly preserving the interconnections among the elements. The target model is usually structured as a tree in which the root element contains other elements by means of the aggregation relation. There can be other references among the nodes of the tree. The generation process in the proposed algorithm (see Line 71) is based on a recursive function that is initially applied to the root. It creates an output element in the rule for each entity of the output model example. The following steps forms this generation process.

1. The root element of the output is selected.
2. The elements contained by the root are recursively added before the creation of the current element, and identified with a particular MTL variable.
3. After that, the current element is created, with its aggregation relations pointing to the MTL variables of the previously created elements. In this way, the process preserves the connections among the target elements.
4. For each created entity, its attributes are filled in with either the values indicated in the output model or information extracted from the input model.

```
71: function GenerateOutputElement(dictionary, e: element): $OutputElements$
72: outElems = $\varnothing$
73: out = $\oslash$
74: for each g $\in$ e.G do
            outElems = outElems $\oplus$ GenerateOutputElement(dictionary, $e.g.e_2$)
75: end for
```

```
76: **for each** a $\in$ e.A **do**
77:         **for each** entry $\in$ dictionary **do**
78:                 value = replace(a.v, entry.e.n, entry.c)
79:                 out = out $\oplus$ **assign(** $\oplus$ a $\oplus$ **,** $\oplus$ value $\oplus$ **)**
80:         **end for**
81: **end for**
82: **for each** g $\in$ e.G **do**
83:         out = out $\oplus$ **assign(** $\oplus$ g $\oplus$ **,** $\oplus$ $e.g.e_2$ $\oplus$ **)**
84: **end for**
85: outElems = outElems $\oplus$ **,** $\oplus$ out
86: **return** outElems
```

3.4 Mapping of Property Values between the Input and Output Models

The algorithm assumes that attributes in the output model may contain expressions referring to values found in elements of the input model. In the output model, the designer refers to information from the input model using the *Transformation IDentifiers* (TIDs) (see Section 3.2). Given this identifier, all its related properties can be obtained from the dictionary. During the generation of the output of the rule, the attributes are set with the correct expression obtained from the corresponding input model element. This expression is usually the path of the corresponding element from the *main* element of the input side of the rule. The constraint construction follows rules [8].

```
87: **procedure** UpdateDictionary (**var** dictionary, e, c)
88: **where** e: element;
89:         c: constraint expression of the element from *main* element
90: $dictionary = dictionary \bigcup \{< e, c >\}$
91: **for each** r $\in$ {e.R $\cup$ e.G} **do**
92:         newConstraint = c $\oplus$ $\rightarrow$ $\oplus$ r.n
93:         UpdateDictionary(dictionary, $e.r.e_2$ , newConstraint)
94: **end for**
```

The MTBE algorithm works for this issue as follows. Firstly, during the generation of the input side of the rule (see Line 7), pairs of TIDs and the necessary constraint expression to access it are inserted in the dictionary with the procedure *UpdateDictionary* (see Line 87). If possible, the constraint expression references the corresponding element using as origin the *main* element. Otherwise, the constraint expression uses the type of the element and the values of its attributes. Secondly, in the generation of the output, the TIDs are replaced with the constraint expressions associated to the corresponding TIDs. This way, if an attribute contains the name of a known TID, it is replaced with a correct constraint expression indicating how to get the information from the input model.

4 Evaluation

For experimentation, this algorithm with a graphical user interface has been implemented with the Java programming language for generating rules defined with ATL [7]. This tool, called *MTGenerator*, with its source code can be downloaded from *Grasia* web [6]. Some MTs generated by this tool have been applied to software developments, and some of these MTs are either described in other publications or available in our web site, as Table 2 presents. In this table, *N:1* and *N:N* denote *one-to-many* and *many-to-many* rules respectively.

Table 2. MTs generated by the MTGenerator tool

	Kinds of rules	Other MTBE tool can generate	Requires further manual adjustment	Published in or available from
UseCase2Agent	N:N	No	No	[4,5]
InitialTaskWorkflow	1:N	Yes	Yes	[4]
NonInitialTaskWorkflow	N:N	No	No	[4]
Task2CodeComponent	1:N	Yes	No	[4]
InteractionUnit	N:N	No	No	[4]
Agent2Deployment	1:N	Yes	No	[4]
Deployment2Testing	1:N	Yes	No	[4]
RefactoringADELFE	N:N	No	Yes	[5]
UseCase2Interaction	N:N	No	No	*Grasia* web [6]
InteractionDefinition2 2InteractionProtocol	N:N	No	No	*Grasia* web [6]
AddSkill	N:N	No	No	*Grasia* web [6]

Considering the eleven MTs of Table 2, seven of these MTs cannot be generated by other existing MTBE tools [17,15] because the other tools cannot generate many-to-many rules. For instance, the generation of these kinds of rules becomes crucial in agent-oriented software engineering, in which designers are not used to model transformation languages and many-to-many transformation rules are necessary as proven in our previous work [4,5].

Furthermore, nine out of the eleven MTs (see Table 2) did not require any manual adjustment after the generation. However, two MTs required manual adjustment because it was necessary to add an additional constraint; for instance, the constraint that was added to the *InitialTaskWorkflow* MT is described in [4]. For preventing this manual adjustment, in the future the algorithm can consider *negative examples* [15], which prevent the execution of a rule in case the input matches with a negative example. The algorithm can incorporate the generation of negative constraints to incorporate the negative examples. A negative example for the *InitialTaskWorkflow* MT is proposed in [4].

Finally, for illustrating the application of the presented algorithm and tool, an example of a the input part of a rule generated by the MTGenerator tool follows. In this piece of ATL code, several input elements are considered by adding constraints related to the *main* element.

```
cin:MMA!UInitiates(
    cin.iniSource.multiplicity=1 and
    cin.iniSource.itRole.id='R1' and
    cin.iniTarget->select(t|
            t.multiplicity=2 and
            t.isInteractionUnit.id='IUA').notEmpty() )
```

A piece of the ATL code of the rule generated by the MTGenerator tool follows. Some elements are produced by the rule and they are connected among them. In addition, the *multiplicity* value of the first element is taken from the input part of the rule.

```
outa1:MMB!InteractionSource(
       multiplicity<-cin.iniSource.multiplicity,
       isInteractionUnit<-oute1),
...
outr1:MMB!UColaborates(
       label<-'',
       colSource<-outa1,
       colTarget<-outa2),
```

A piece of ATL code of the imperative part of the rule generated by the MT-Generator tool follows. In this piece of code, an element created in the output part of the rule is inserted in auxiliary variables, and then these elements will be allocated in the output model.

```
thisModule.newRelations
    <-thisModule.newRelations.append(outr1);
```

5 Related Work

The key works in MTBE are those of Wimmer et al. [17] and Varro and Balogh [13,15], who presented at almost the same time implementations of this approach. This section discusses the differences between these works and the algorithm introduced in this paper. Table 3 summarizes the most relevant features of each approach.

Wimmer et al. [17] presents the generation of *Model Transformations By Example* and named it as MTBE. They present MTBE as the automated generation of transformations from examples of the input and output models. This avoids the user knowing neither the transformation language, the metamodels, nor even the abstract syntax of the modeling languages. The work is illustrated with the ATL language.

The work of Varro and Balogh [13,15] uses inductive logic programming to derive the transformation rules with a MTBE approach. An innovation of that work is the learning of negative constraints from negative examples. This work also introduces the *connective analysis* to consider the references among modeling elements. However, this analysis is restricted to the rule outputs. The networks of modeling elements in rule inputs are not considered.

Table 3. Comparison of the features of existing MTBE approaches

Features of MTBE	Varro and Balogh	Wimmer et al	Our Technique
Mapping of attributes	yes	yes	yes
Propagation of links	yes	yes	yes
Negative Examples	yes	no	no
Generation of Constraints	no	yes	yes
Explicit Allocation of Target Elements	yes	no	yes
Limit number of input elements of rules	1	1	no-limit
Limit number of output elements of rules	1	1	no-limit

In both cases, the main difference with the work in this paper is the generation of many-to-many rules. Wimmer et al. only generates ATL rules that transform one element into another . Varro and Balogh's also generate rules for only one input element, although the output of a transformation can be a graph by means of the connective analysis. On the contrary, our approach is able to generate many-to-many rules by means of OCL constraints within the input side of the rules. This allows one to transform networks of modeling elements into other networks, and to match information from groups of input elements into groups of output elements.

6 Conclusions and Future Work

This paper presents an algorithm for the generation of transformations with a MTBE approach and its implementation for ATL. The main contribution of the algorithm is the processing of rules with multiple graphs of elements as input using embedded mappings. The tool implementing this algorithm is publicly available from *Grasia* web [6].

As a future line of research, the algorithm can consider negative examples, which can be added to the pairs of examples. Then, each rule can ge generated from a positive source example, several negative source examples and a target example. The negative examples will be translated into negative constraints in the rule. In this manner, more complex model transformations can be defined without manual adjustment.

Acknowledgements. This work is supported by the project *Agent-based Modeling and Simulation of Complex Social Systems* (SiCoSSys), funded by Spanish Council for Science and Innovation, with grant TIN2008-06464-C03-01.

References

1. Allilaire, F., Bézivin, J., Jouault, F., Kurtev, I.: ATL-eclipse support for model transformation. In: Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France (2006)

2. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modelling Framework: Developer's Guide. Addison Wesley, Reading (2003)
3. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
4. García-Magariño, I., Gómez-Sanz, J.J., Fuentes-Fernández., R.: INGENIAS Development Assisted with Model Transformation By-Example: A Practical Case. In: 7th International Conference on Practical Applications of Agents and Multi-Agent Systems, PAAMS 2009 (2009) (papers section), http://grasia.fdi.ucm.es
5. García-Magariño, I., Rougemaille, S., Fuentes-Fernández, R., Migeon, F., Gleizes, M.-P., Gómez-Sanz, J.J.: A Tool for Generating Model Transformations By-Example in Multi-Agent Systems. In: 7th International Conference on Practical Applications of Agents and Multi-Agent Systems, PAAMS 2009 (2009) (papers section), http://grasia.fdi.ucm.es
6. Grasia web: http://grasia.fdi.ucm.es (in Software → MTGenerator section)
7. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
8. Krishnamurthi, S., Gray, K.E., Graunke, P.T.: Transformation-by-Example for XML. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, p. 249. Springer, Heidelberg (2000)
9. Lawley, M., Steel, J.: Practical Declarative Model Transformation with Tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
10. OMG. Meta Object Facility(MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification. Object Management Group (05-11-01) (November 2005)
11. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
12. Siikarla, M., Laitkorpi, M., Selonen, P., Systä, T.: Transformations Have to be Developed ReST Assured. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 1–15. Springer, Heidelberg (2008)
13. Varro, D.: Model transformation by example. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)
14. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3), 187–207 (2007)
15. Varró, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: Proceedings of the 2007 ACM symposium on Applied computing, pp. 978–984 (2007)
16. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional, Reading (2003)
17. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation By-Example. In: Proceedings of the 40th Annual Hawaii International Conference on System Sciences, vol. 40(10), p. 4770 (2007)

A Collection Operator for Graph Transformation

Roy Grønmo[1,2], Stein Krogdahl[1], and Birger Møller-Pedersen[1]

[1] Department of Informatics, University of Oslo, Norway
[2] SINTEF Information and Communication Technology, Oslo, Norway
{roygr,birger,steinkr}@ifi.uio.no

Abstract. Graph transformation has a well-established theory and associated tools that can be used to perform model transformations. However, the lack of a construct to match and transform collections of similar subgraphs makes graph transformation complex or even impractical to use in a number of transformation cases. This is addressed in this paper, by defining a collection operator which is powerful, yet fairly simple to model and understand. We present model transformation examples from different modeling domains to illustrate the benefit of the approach.

1 Introduction

Graph transformations have been proposed by several authors as a means to perform model transformations [4,7]. The graphical way to define graph transformations, the available tool support [10,23,25], and the well-established theory including termination and confluence analysis [16,21] makes graph transformation appealing.

The graph concept is based on nodes and directed edges from which we can define models. Many model transformations can then be defined by a set of graph transformation rules, where each rule consists of a left hand side (LHS) graph, a right hand side (RHS) graph, and an interface (I) graph. The elements in the interface graph are to be preserved, the elements in LHS $\setminus$ I are to be deleted, and the elements in RHS $\setminus$ I are to be added.

The minimalistic nature of graph transformation is probably a key factor to its success, since it makes it easier to implement tools and to establish theory on its concepts. For the graph transformation designer, on the other hand, the lack of higher level constructs reduces the usability of graph transformation. This is why some authors have proposed to raise the level of abstraction by introducing new and powerful graph transformation mechanisms, e.g. *the star operator* [17] and *recursion* [12]. Our experience on a number of graph transformation examples reveals an often occurring need to match collections of similar subgraphs, which is addressed by our *collection operator*. The collection operator allows us to express a fairly powerful model transformation using a single rule.

Fujaba [10] and PROGRES [23] have support for matching collections of single nodes only. In many cases this is too restrictive and a lot of recent approaches [1,5,9,14,18,22] address this by allowing to match collections of subgraphs. Our collection operator aims to be concise and easy to use for the rule designer, and at the same time expressive enough for many typical model transformation scenarios.

The paper is structured as follows. Section 2 provides the formal foundation of graph transformation; Section 3 presents the collection operator; Section 4 shows how complicated it is to simulate a rule with collection operators by multiple collection free rules in the AGG graph transformation tool; Section 5 shows three example rules with collection operators; Section 6 covers related work; and Section 7 concludes.

2 Graph Transformation

Below we provide the known formal foundation of graph transformation [15].

Definition 1 (Graph and graph morphism). *A* graph *$G = (G_N, G_E, src, trg)$ consists of a set G_N of nodes, a set G_E of edges, two mappings $src, trg : G_E \rightarrow G_N$, assigning to each edge $e \in G_E$ a source node $src(e) \in G_N$ and target node $trg(e) \in G_N$. A* graph morphism *$f : G_1 \rightarrow G_2$ from one graph to another, with $G_i = (G_{E,i}, G_{N,i}, src_i, trg_i), (i = 1, 2)$, is a pair $f = (f_E : G_{E,1} \rightarrow G_{E,2},\ f_N : G_{N,1} \rightarrow G_{N,2})$ of mappings, such that $f_N \circ src_1 = src_2 \circ f_E$ and $f_N \circ trg_1 = trg_2 \circ f_E$ (preserve source and target).*

A graph morphism $f : G_1 \rightarrow G_2$ is injective if f_N and f_E are injective mappings. Only injective graph morphisms will be relevant in this paper.

Definition 2 (Rule). *A graph transformation rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ consists of three graphs L(LHS), I(Interface) and R(RHS) and a pair of injective graph morphisms $l : I \rightarrow L$ and $r : I \rightarrow R$.*

Definition 3 (Match). *Given a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ and a graph G. Then an occurrence of L in G, i.e. an injective graph morphism $m : L \rightarrow G$, is called* match. *The function* isMatch *$: L, G, (L \rightarrow G) \rightarrow Bool$ returns true if and only if $L \rightarrow G$ is a match from L to G. A match m for rule p satisfies* the dangling condition *if no node in $m(L \setminus l(I))$ is incident to an edge in $G \setminus m(L \setminus l(I))$.*

Definition 4 (Derivation Step). *Given a graph G, a graph transformation rule $p : L \xleftarrow{l} I \xrightarrow{r} R$, and a match $m : L \rightarrow G$, then there exists a* derivation step *from the graph G to the graph H if and only if the dangling condition is satisfied. H is constructed as follows:*

1. *Remove the image of the non-interface elements of L in G, i.e. $H' = G \setminus m(L \setminus l(I))$.*
2. *Add the non-interface elements of R into H, i.e. $H = H' \cup (R \setminus r(I))$.*

A *negative application condition* [15] is an extension of the LHS which prevents matches from being applied in a derivation step.

Definition 5 (Negative application condition (NAC)). *A NAC for a graph transformation rule $L \xleftarrow{l} I \xrightarrow{r} R$, is defined by a pair of injective graph morphisms:*

$L \xleftarrow{s} NI \xrightarrow{t} N$, *where* N *is* the negative graph, *and* NI *defines the interface graph between* L *and* N. *A match* $m : L \to G$ *satisfies the NAC if and only if there does not exist an injective graph morphism* $n : N \to G$ *which preserves the NI interface mappings, i.e. for all nodes* v *in* NI *we have* $n_N(t_N(v)) = m_N(s_N(v))$ *and for all edges* e *in* NI *we have* $n_E(t_E(v)) = m_E(s_E(e))$. *A rule can have an arbitrary number of NACs, and a derivation step can only be applied if a match satisfies all the NACs of the matched rule.*

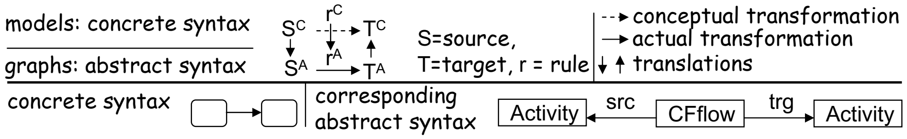
In addition to the above, we adopt the theory of *typed attributed graphs* [13], where graphs are extended by assigning types to nodes and edges, and by assigning a set of named attributes to each node type. A graph morphism must now also preserve the node and edge types, and the attribute values.

In the graph transformation rules throughout this paper we only explicitly display the LHS and the RHS graphs, while the interface graph is given by shared identifiers of elements in the LHS and the RHS. Such identifiers are displayed next to its element.

2.1 Concrete and Abstract Syntaxes

Typed attributed graphs are rich enough to represent most of todays modeling languages in a natural way. These graphs use a generic graphical layout, called *abstract syntax*, where nodes are visualized as rectangles containing the type name and a list of attribute values, and edges are visualized as directed arrows with the type name. *The concrete syntax* of a modeling language uses a tailored visualization with icons and rendering rules depending on the element types. To improve the usability for the graph transformation designer, we define the transformation rules upon concrete syntax. The transformation designer can think entirely in the concrete syntax, while the matching and derivation steps are still carried out in the abstract syntax.

In a natural translation of UML activity models [19] to typed attributed graphs, an activity in the concrete syntax corresponds to a node of type `Activity` in the abstract syntax. A control flow in the concrete syntax corresponds to a node of type `CFlow` and two edges of types `src` and `trg`. The figure below shows an activity model concrete syntax on the left and the corresponding abstract syntax on the right.



Our examples use the concrete syntax, while the formalization is defined on the abstract syntax. We assume that the translation from concrete to abstract syntax (`c2a`), and the opposite direction (`a2c`), is already defined for the relevant modeling languages. Then we can link concrete syntax-based graph transformation to abstract (and traditional) syntax-based graph transformation in a systematic way: 1) translate the concrete syntaxes of source model and rules (consisting of L, I, R, NI, and N models) into abstract syntax graphs by `c2a`, 2) apply the abstract syntax graph transformation rules on the source graph, and 3) translate the resulting abstract syntax graph to a concrete model by `a2c`.

Linking concrete syntax-based graph transformation to the traditional graph transformation has been successfully applied in our previous work [11] and by other authors [3,26]. With a large number of modeling languages, including those illustrated in this paper, the same translation (`c2a`) is reasonable to use for both the rules and the source model. When the same translation is used for both the rules and the source model, the principles of graph transformation can be directly applied at the concrete syntax level, and the transformation designer does not have to care about the underlying translations to graphs at the lower abstraction level.

3 The Collection Operator

We propose a collection operator that can be used in a graph transformation rule to match and transform a set of similar subgraphs in one step. Figure 1 illustrates the collection operator in a workflow refactoring example [6,11]. The source model (labeled 1) is an activity model with two consecutive decision nodes (displayed as diamond symbols), and two inner paths leading to the activities named `doA` and `doB`. The refactored model (labeled 2) shows that the two decision nodes can be combined into one.

Since there can be an arbitrary number of inner paths, plain graph transformation as defined above cannot express the removal of a redundant decision node with

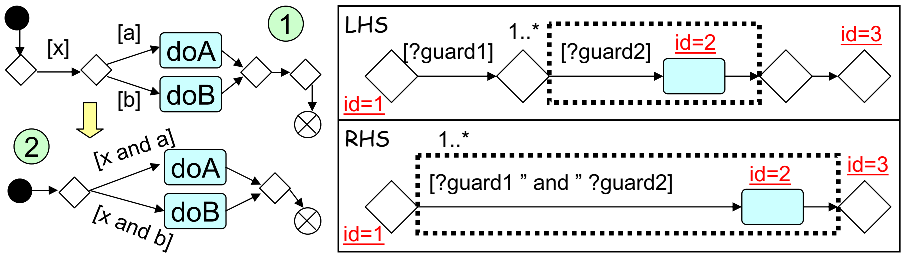


Fig. 1. Activity model refactoring: Removing redundant decision node

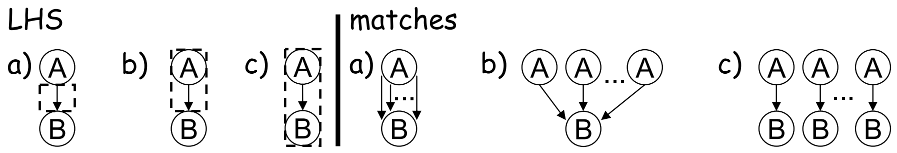


Fig. 2. Semantics of the collection operator

a single rule. In the right part of Figure 1 a single rule with the collection operator (dashed line frame) is sufficient to do the refactoring. The collection operator matches an arbitrary number of subgraphs, which all have an inner path leading to a single activity node between the inner decision and merge nodes. The outer guard (`?guard1`) is combined with each inner guard (`?guard2`) by using `and` operators. Notice that a matched subgraph at the abstract syntax level contains three nodes (one node of type Activity and two nodes of type CFlow) and associated edges.

The size of the collection match must be greater or equal to the lower bound cardinality (`1` in the example) in order to apply a rule. A collection match size is non-deterministically increased until we reach the upper bound (no limit in the example) or there are no more possible subgraph matches. The parts outside of the collection operator must occur only once in a rule match.

Identifiers (e.g. id=1) are associated to the main elements such as activities, control nodes and control flow. Attribute variables (e.g. `?guard2`) are associated with the values of attributes such as activity name and guard. An identifier/variable inside a collection represents a set of identifiers/variables.

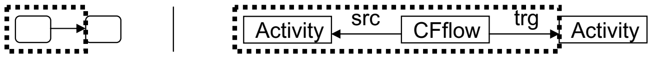
In Figure 2 we use a simple concrete syntax of named circles connected by arrows to show the relationship between the collection operator in the LHS and possible matches. In case a) only the arrow is inside the collection while the source and target circles are outside the collection. This means that possible matches have a set of arrows between the same two circles. In case b) the circle named `A` is also inside the collection which means that a match contains a set of distinct `A` circles with arrows leading to the same target circle named `B`. In case c) both the circles are inside the collection which means that a match contains a set of distinct circles each having their own arrow.

A NAC and the RHS can only use collection operators that are introduced in the LHS. The RHS indicates the changes to each subgraph match in the collection, and the cardinality of the same collection operator must be the same in the LHS and the RHS/NACs. The actual matching collection size of the LHS leads to the same collection instantiation size within the RHS/NACs. If the collection operator is absent in the RHS, then it implies a deletion of all the matched collection subgraphs.

A collection operator has an identifier which is visualized next to the collection frame. No collection identifier visualization is needed in cases where the collection operator is uniquely identified by its cardinality, or when the rule has only one collection operator (e.g. Figure 1). To avoid complexity we disallow collection operators to be overlapping or nested in the abstract syntax, which also implies that two collection operators cannot be adjacent in the concrete syntax. Otherwise the two collection operators would include at least one common edge in the abstract syntax. This fact becomes clearer in the following subsection.

3.1 Mapping Collection Operator from Concrete to Abstract Syntax

In the translation from concrete syntax rules to abstract syntax rules, we must determine which abstract syntax elements belong to the collection. This is illustrated in the figure below with a collection operator in concrete syntax to the left, and corresponding abstract syntax to the right:



If an element is inside a collection in the concrete syntax, then a corresponding node goes inside the collection in the abstract syntax (e.g. leftmost `Activity` node and `CFlow` node). An edge connecting two nodes that are both inside a collection, belongs to the collection (e.g. `src` edge).

An edge in the abstract syntax connecting a collection node to a non-collection node must also be included in the collection (e.g. `trg` edge). This is because all edges shall have exactly one source and one target node (note: source and target should not be confused with the example edges of type `src` and `trg`). Otherwise, with a non-collection edge incident to a collection node, the only possible collection cardinality is 1..1, which implies that the collection is redundant.

3.2 Collection Operator Formalized

A collection operator can be represented in graphs as a node of type `coll`, with `min` and `max` as cardinality attributes, and with a set of edges targeting all the collection subgraph nodes. The set of all collection operators in a rule p is referred to as $Coll_p$. We use ψ to denote a function that maps each collection operator in a rule p, to a number within its cardinality range, i.e. $\psi : Coll_p \rightarrow (\mathbf{N} = \{0, 1, 2, \ldots\})$, where $\forall c \in Coll_p : \psi(c) \in [c.min, c.max]$.

For a rule $p : L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ with at least one collection operator, we let $p^\psi : L^\psi \overset{l}{\leftarrow} I^\psi \overset{r}{\rightarrow} R^\psi$ denote the collection free rule where all collection operators in p are replaced by the ψ mapped number of collection content copies. In these copies all the copied elements/attributes get fresh identifiers/variables respectively, while the interface elements between the LHS and the RHS are maintained. Similarly, $L^\psi \overset{s}{\leftarrow} NI^\psi \overset{t}{\rightarrow} N^\psi$ denotes a collection free NAC.

Figure 3 shows p^ψ, where c_1 is the collection operator in the transformation rule p representing the redundant decision node example from Figure 1, and $\psi(c_1) = 2$.

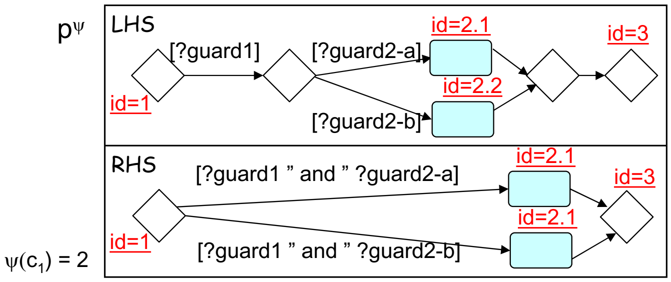


Fig. 3. The rule for activity model refactoring with 2 as the collection size

Definition 6 (Extensions). *Given a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ with at least one collection and a graph G. A collection cardinality mapping ψ^+ extends the cardinality mapping ψ (denoted $\psi^+ \succ_p \psi$) if and only if there is at least one greater collection cardinality and none of the collection cardinalities are smaller:*

$$\psi^+ \succ_p \psi \overset{\mathsf{def}}{=}$$
$$\exists c \in Coll_p : \psi^+(c) > \psi(c) \;\wedge\; \forall c \in Coll_p : \psi^+(c) \geq \psi(c)$$

A rule p^{ψ^+} extends the rule p^{ψ} (denoted $p^{\psi^+} \sqsupset p^{\psi}$) if and only if the following holds: $\psi^+ \succ_p \psi$ and the three graphs L^{ψ^+}, I^{ψ^+}, and R^{ψ^+} contains respectively L^{ψ}, I^{ψ}, and R^{ψ} as subgraphs.

An injective morphism $m^{\psi^+} : L^{\psi^+} \to G$ extends the injective morphism $m^{\psi} : L^{\psi} \to G$ (denoted $m^{\psi^+} \sqsupset m^{\psi}$) if and only if $p^{\psi^+} \sqsupset p^{\psi}$ and $m^{\psi^+}(L^{\psi}) = m^{\psi}(L^{\psi})$.

Definition 7 (Match for a rule with collections (cMatch)). *Given a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ with at least one collection, a graph G, and a collection cardinality mapping ψ. An injective morphism $m^{\psi} : L^{\psi} \to G$ is a cMatch of rule p in G if and only if m^{ψ} is a non-extendable injective morphism in G. Formally,*

$$isCMatch(L, G, \psi, m^{\psi}, L^{\psi}) \overset{\mathsf{def}}{=}$$
$$isMatch(L^{\psi}, G, m^{\psi}) \wedge$$
$$\nexists m^{\psi^+} \in (L^{\psi^+} \to G) : \; (m^{\psi^+} \sqsupset m^{\psi}) \;\wedge\; isMatch(L^{\psi^+}, G, m^{\psi^+})$$

When we have a cMatch $m^{\psi} : L^{\psi} \to G$ for a rule p with collections, then m^{ψ} is also a match in the collection free rule $p^{\psi} : L^{\psi} \xleftarrow{l} I^{\psi} \xrightarrow{r} R^{\psi}$ where Def. 4 for derivation steps is still valid. We also get collection free NAC definitions as $L^{\psi} \xleftarrow{s} NI^{\psi} \xrightarrow{t} N^{\psi}$, where Def. 5 applies.

3.3 Inherent Tool Support for Rules with Collection Operators

This section describes how we can provide tool support for rules with collection operators. The minimal configuration of ψ for which we can find a cMatch for a rule p with collection operators, is when $\forall c \in Coll_p : \psi(c) = c.min$. We refer to this minimal configuration of ψ as ψ^-. Given a rule p with collection operators and a graph G, the following steps can be used to find a cMatch in p and try to apply a derivation step for that cMatch:

1. Look for an injective morphism $m^{\psi^-} : L^{\psi^-} \to G$ in the collection free rule p^{ψ^-}.
2. Extend (if possible) the injective morphism m^{ψ^-} until it is a non-extendable injective morphism $m^{\psi} : L^{\psi} \to G$, i.e. a cMatch for p. The extension process can be achieved by iterating over each collection operator $c \in Coll_p$ and increasing $\psi(c)$ as much as possible. $\psi(c)$ can only be increased by 1, if the injective morphism can be extended with an additional subgraph match of the collection content in c.

3. Apply a derivation step with the collection free rule p^ψ and the match m^ψ if m^ψ satisfies all the NACs and the dangling condition.

We use a transformation task of state machine refactoring [24] to illustrate the proposed matching process above. The refactoring applies to cases where all the inner states of a composite state have outgoing transitions to the same state, and all these outgoing transitions share the same trigger and effect, while the guards must all be undefined or equivalent. In such cases we can replace all these outgoing transitions by a single transition from the composite state to the external state.

The top left part of Figure 4 shows an example state machine modeling the behavior of a smartphone (based on [4]). The state called `Idle` represents a waiting state of a smartphone. The signal `phoneMode` triggers a composite state named `Active` in which we can make phone calls. All the inner states have a trigger with the same outgoing trigger `hangUp` targeting the outer `Idle` state.

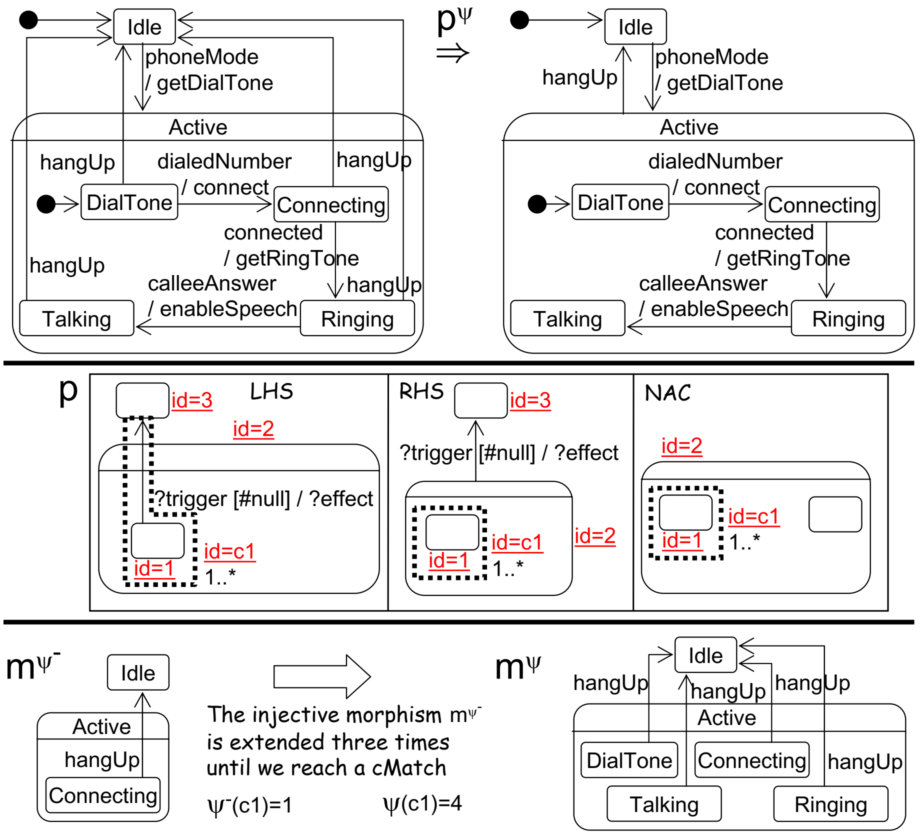


Fig. 4. State machine refactoring: phone example (top), transformation rule (middle), matching process (bottom)

The middle part of Figure 4 shows a transformation rule, named p, that defines the refactoring. The rule uses a collection operator with the id `c1`, where a transition is inside the collection and its three attributes are outside of the collection. Recall that the parts outside of the collections occur once, and must therefore have the same value for all the transitions. When the variables or values must be shared by collection nodes, we call them *shared variables* (e.g. `?trigger` and `?effect`) or *shared values* (e.g. `#null`). We have introduced a keyword `#null` to indicate that all the guard values shall be undefined (the same interpretation as a `true` value).

We have included a NAC to ensure that all the substates within the composite state have the requested transitions to the external state. The matching is injective, which means that the NAC prohibits the existence of other substates than those already matched by the LHS and repeated with $\underline{\text{id=1}}$ in the NAC. The new transition in the RHS gets the same trigger and effect values as those shared by all the replaced transitions, and it gets an undefined guard value.

The bottom part of Figure 4 illustrates how the matching algorithm works. First we non-deterministically find a match m^{ψ^-} of the rule m^{ψ^-}, which is an injective morphism for the rule p (shown in the bottom left part). The injective morphism m^{ψ^-} is extended by three subgraph matches of the collection content until we reach the cMatch m^{ψ}. The top right part of the Figure 4 shows the refactored model after applying the match m^{ψ} and the rule p^{ψ} on the source model.

4 Simulating a Collection Rule by a Transactional Sequence of Collection Free Rules

As an alternative to implementing the new matching algorithm from Section 3.3, this section shows how a rule with collection operators can be simulated in an existing graph transformation tool such as AGG [25]. Since the collection operator is not available, we use a transactional sequence of multiple collection free rules to simulate the intended effects of a single rule with collection operators. An early prototype of the approach was described in [11]. We only consider NAC free rules in this section.

The complicated apparatus and the set of less intuitive collection free rules show the large benefits for the transformation designer to have direct support for the collection operator. The alternative is to manually define and ensure a correct execution strategy of collection free rules, which is time consuming and error prone.

A rule r with collection operators can be represented by zero or one Init rule, one or more Iter rules and zero or one Final rule. These rules are ordered and executed as a transaction.

Init rule. The rule shall be applied only once as the first rule in the transaction. This rule has $LHS = L^{\psi^-}$, which ensures that there is a match of the original rule r. The RHS copies the LHS and adds all (if any) of the to-be-added non-collection elements. It must be the first applied rule, since the other rules may connect to the added elements from this rule. **Iter rules**. Each collection operator is mapped to an Iter rule. The Iter rule shall be applied to each subgraph match of a collection. **Final rule**. The Final rule deletes all non-collection elements.

The transformation of a rule r with collection operators is defined by the pseudocode of algorithm 1. The `numCollections` method returns the number of collection operators. The `remColls` method removes all collection operators including their content. The `collContent(i)` method retrieves the content inside collection operator number `i`. $Iter_i$ is the rule for collection i. Rule $Iter_i$ only applies the changes relevant to collection i. By not changing any other parts, all the individual matches within collection i as well as the other collections get an equal chance to be matched.

Algorithm 1. ToCollFree($r : CollRule$)

```
$nonCollAdd = r.R.remColls \setminus r.I.remColls$
$Init =$ new $Rule$; $Init.L = r.L^{\psi^-}$; $Init.R = Init.L \cup nonCollAdd$
for $i \leftarrow 1$ to $r.numCollections$
    do $Iter_i =$ new $Rule$
       $Iter_i.L = r.L.remColls \cup nonCollAdd \cup r.L.collContent(i)$
       $Iter_i.R = r.L.remColls \cup nonCollAdd \cup r.R.collContent(i)$
if $(r.L.remColls \setminus r.I.remColls) <> \emptyset$
    then $Final =$ new $Rule$;    $Final.L = r.L.remColls$
         $Final.R = r.R.remColls \setminus nonCollAdd$
```

Figure 1 showed an example of a rule with collection operators. By following algorithm 1 we get a set of rules, $\{Init, Iter, Final\}$, as shown in Figure 5. The Final rule is produced since there are non-collection elements to-be-deleted. The Iter rule replaces a path of control flows going to the innermost decision and merge nodes, with a new path of control flows only going to the outermost decision and merge nodes with a combined guard. The Iter rule replaces one path each time the rule is applied. The Final rule is applied when the Iter rule is no longer applicable.

We need to ensure that all the rules in a single transaction involve the same context regarding the non-collection elements, which we achieve by introducing an additional `id` attribute for all the elements. All the non-collection elements in the Iter and Final rules gets id values corresponding to the elements matched by the Init rule.

Conceptually, the collection rule LHS builds an entire match, and then applies the effect defined by the RHS. When simulating such a behavior with multiple rules in AGG, we need to be careful about possible dependencies and interactions between the Iter rules. One Iter rule may for instance add elements leading to yet another individual matching of another Iter rule, which is incorrect behavior. To avoid this problem we extend all the model elements by a boolean helper attribute named `exclude`. All `exclude` attributes are set to `false` at the start

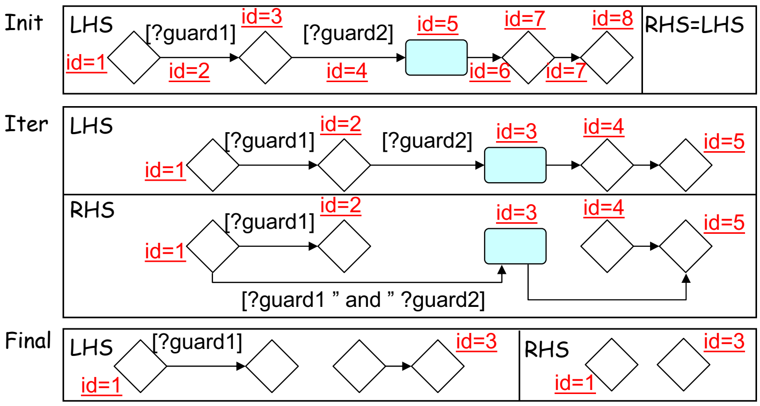


Fig. 5. Activity model refactoring (collection free)

of the transaction, while all the collection content `exclude` attributes are set to `true` in the RHS of the Iter rules. Furthermore, the LHS of the Iter rules are extended so that they only match elements with the `exclude` attribute set to `false`. By doing so, the Iter rules can be applied in an arbitrary order.

In other tools with more control flow and transaction support like in Fujaba [10] and PROGRES [23], it may be simpler than with AGG to simulate a collection rule by collection free rules. Still, the introduction of a collection operator will greatly reduce the effort needed by the transformation designer when designing rules.

5 Examples

In this section we show three examples where the collection operator is helpful.

Fire transition in petri nets. A petri net model consists of *places*, *transitions* and directed arrows. The directed arrows goes from a place to a transition or from a transition to a place. A transition T_1 has a *preset* of places which is the places that have a directed edge to T_1, and T_1 has a *postset* of places which is the places that have a directed edge from T_1. At any moment a number of *tokens* are assigned to each place, and each token is assigned to exactly one place.

In our concrete syntax, the tokens are drawn as small, filled circles, places are drawn as larger, unfilled circles, and transitions are drawn as rectangles. An example is shown at the top left of Figure 6 with label 1, where we have a single transition consisting of two places in the preset and three places in the postset. The places in the preset have one and two tokens respectively. The places in the postset have zero, zero and one token respectively.

A transition is *enabled* when all the places in the preset of a transition have at least one token. The transition, within the model labeled 1 in Figure 6, is thus enabled and we can *fire* a transition. When firing a transition we shall remove

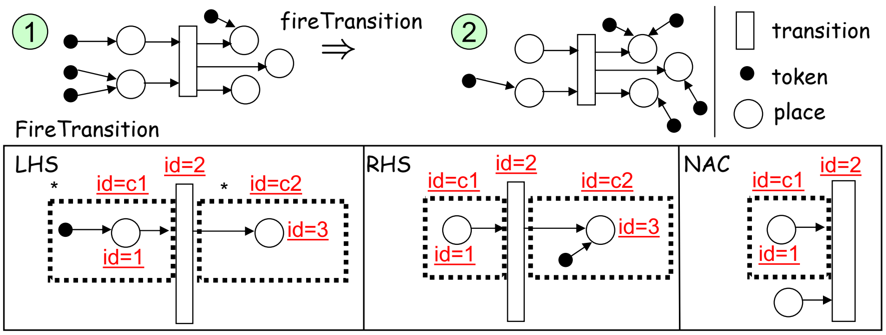


Fig. 6. *Top*: The effects of firing a transition on a petri net model, *bottom*: fire transition rule

one token from each place in the preset and add one token to each place in the postset. The resulting model after firing the transition is shown with label 2.

With two collection operators (identified as $c1$ and $c2$) we can define the firing of a transition by a single rule. Collection $c1$ expresses that we remove one token from each place in the preset, while collection $c2$ expresses that we add one token to each place in the postset. The NAC ensures that there are no preset places without a token.

Activity model refactoring: add fork. UML activity models allow an activity to have multiple outgoing control flows, which are interpreted as an implicit fork. It is normally encouraged to use an explicit fork node instead, which we can automatically introduce by the leftmost rule in Figure 7. We assume that the rule editor is more flexible than typical activity model editors, by allowing a control flow without a target. The missing target allows any kind of target node type in the model match. If a target node is required in the editor, then we can use an abstract supertype from the UML metamodel representing the possible target nodes. This type will be displayed with the abstract syntax as the target node in the rule. The lower cardinality of the collection operator is 2, so that the fork node is only introduced when there is more than one outgoing control flow.

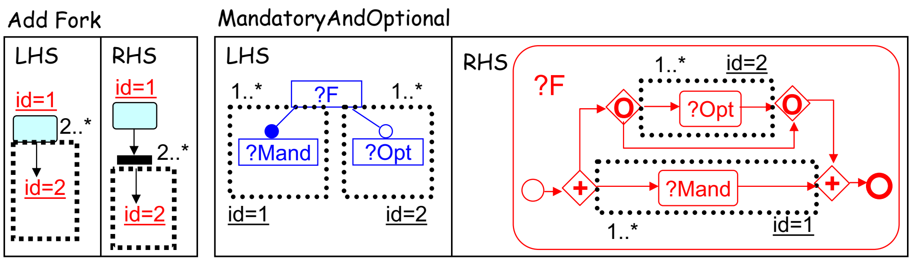


Fig. 7. *Left*: Activity model refactoring, *Right*: From feature models to BPMN

From feature models to BPMN. This example, given by the rightmost rule in Figure 7, shows a need for two collection operators in the same rule. The rule `MandatoryAndOptional` is one of many rules we have defined to transform from feature models [2] to Business Process Modeling Notation (BPMN) [20] (BPMN models are very close to UML 2 activity models). For this example transformation, the sibling features are assumed to represent independent tasks. The rule is simplified compared to the complete rule that works recursively when the child features themselves also are parent features.

Features are mapped to BPMN activities. Activities of child features are placed inside independent control flow branches of an activity. We use two collection operators, one for optional tasks and the other for mandatory tasks.

A parent feature with the arbitrary name `?F` is mapped to an activity node with the same name. We get an internal `fork-join` branch (fork and join are displayed with a diamond symbol with a plus sign inside) to represent all the mandatory tasks, and an internal `inclusive decision-merge` branch (decision and merge are displayed with a diamond symbol with a circle inside) to represent all the optional tasks.

6 Related Work

In this section we describe related approaches, and these can be distinguished as two groups: 1) collection matching and transformation that is restricted to single nodes only, and 2) collection matching and transformation of subgraphs.

Fujaba [10] and PROGRES [23] have support for matching collections of single nodes only (*set nodes* in PROGRES, *multi objects* in Fujaba), which is a limited expressiveness compared to the collection operator that allows for collections of a fixed but arbitrarily large subgraph. Furthermore, the single node approaches are only defined for abstract syntax. To determine if single node collections are expressive enough for a particular transformation task may depend on the choice of abstract syntax representation of the involved source and target languages.

As an example, we now consider if we can use single node collections to express a rule for firing of petri nets (Figure 6). If the abstract syntax of petri net graph representation uses two different node types to represent tokens and places, then a rule to perform transition firing with single node collections will fail. This is because all tokens of the places in the transition preset will be consumed, and not only one token per place as required. This problem can be avoided by choosing a different abstract syntax where a place has an integer attribute to keep track of the number of tokens instead of having a separate node type for a token. In general it is undesirable to adjust the abstract syntax due to limitations in the rule language. By using *E-graphs* [8] where edges can have attributes we can get away with using single node collections for some of the paper examples, depending on the choice of abstract syntax.

The remaining approaches in this section are all capable of handling subgraph collection matching and transformation. A *group operator*, introduced by Balasubramanian et al. [1] and implemented in the GREaT tool, enables arbitrarily large subgraph matches that can be copied, moved or deleted. However, the subgraph matches can not be modified as with the collection operator.

Amalgamated rules [5] by Jaramillo et al. can simulate the collection operator. Our collection operator is more concise since we can use a single rule, while they need one *subrule* to capture the rule part outside of all collections, and one *elementary rule* for each collection operator.

Nested quantification is proposed by Rensink [22] as an extension to the GROOVE tool, which is similarly concise as our collection operator by allowing a single rule to express subgraph matches. His notation is a bit different from ours since they use exists (∃) and for all (∀) quantifiers to express the parts outside of a collection, and those inside a collection respectively.

Fuss and Tuttlies [9] propose an extension to PROGRES called *set-regions*, which is quite similar to our collection operator. However the concrete notation of such set-regions within the rules is not shown. A strength compared to many other approaches is that they allow for nested set-regions. In this paper we have not allowed the collection operators to be nested, but this seems to be an appropriate extension which we plan to describe in future work.

Minas and Hoffmann [14,18] define a *cloning operator* which is an alternative to our collection operator. Cloned nodes and incident edges correspond to elements inside a collection operator. They support multiple elements inside the same collection operator by assigning the same cloning identifier to several cloned nodes (the incident edges of the cloned nodes implicitly belongs to the same collection).

To our best knowledge none of the other subgraph collection matching approaches have support for shared variables nor collection cardinalities beyond $0..*$ and $1..*$. Furthermore, the other approaches focus only on applying their collection operators for the abstract syntax. The notations by Rensink [22] and as sketched by Fuss and Tuttlies [9] have a nature which makes them appropriate to be introduced on the concrete syntax, which is not the case for Minas and Hoffmann [14,18]. We extend our earlier work [11] where the collection operator was restricted to activity model transformations. The improvements in this paper includes support for multiple collection operators of arbitrary cardinalities in the same rule.

7 Conclusions

In this paper we have introduced the collection operator, which makes graph transformation suitable to use on a number of model transformation cases where it would be cumbersome or impractical without. The collection operator raises the level of abstraction, which is a benefit to the transformation designer. For model transformations where the collection operator naturally applies, Section 4 shows that it is a complicated and time consuming task to manually define transformations without the collection operator.

The collection operator can be used both on the concrete syntax of the modeling language and at the abstract syntax of graphs. A straightforward implementation strategy, described in Section 3.3, shows how we can identify matches and apply derivation steps by reusing much of the existing graph transformation apparatus.

We leave it as future work to investigate how the use of collection operators affect the theory of termination and confluence. It is also future work to decide the conditions under which graph transformation is naturally applicable at the concrete syntax level.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the DiVA project grant no. 215412 (EU FP7 STREP).

References

1. Balasubramanian, D., Narayanan, A., Neema, S., Shi, F., Thibodeaux, R., Karsai, G.: A Subgraph Operator for Graph Transformation Languages. In: ECEASST, vol. 6 (2007)
2. Batory, D.S.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
3. Biermann, E., Ermel, C., Hurrelmann, J., Ehrig, K.: Flexible visualization of automatic simulation based on structured graph transformation. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC (2008)
4. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301. Springer, Heidelberg (2008)
5. de Lara Jaramillo, J., Ermel, C., Taentzer, G., Ehrig, K.: Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets. Electr. Notes Theor. Comput. Sci. 109, 17–29 (2004)
6. Eder, J., Gruber, W., Pichler, H.: Transforming Workflow Graphs. In: Conf. on Interoperability of Enterprise Software and Applications (2005)
7. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
8. Ehrig, H., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graph Transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
9. Fuss, C., Tuttlies, V.E.: Simulating Set-Valued Transformations with Algorithmic Graph Transformation Languages. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 442–455. Springer, Heidelberg (2008)
10. Geiger, L., Zündorf, A.: Tool Modeling with Fujaba. Electr. Notes Theor. Comput. Sci. 148(1) (2006)
11. Grønmo, R., Møller-Pedersen, B.: Aspect Diagrams for UML Activity Models. In: Applications of Graph Transformation with Industrial Relevance (2008)
12. Guerra, E., de Lara, J.: Adding Recursion to Graph Transformation. ECEASST 6 (2007)
13. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505. Springer, Heidelberg (2002)

14. Hoffmann, B., Janssens, D., Eetvelde, N.V.: Cloning and Expanding Graph Transformation Rules for Refactoring. Electr. Notes Theor. Comput. Sci. 152, 53–67 (2006)
15. Lambers, L., Ehrig, H., Orejas, F.: Conflict Detection for Graph Transformation with Negative Application Conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 61–76. Springer, Heidelberg (2006)
16. Levendovszky, T., Prange, U., Ehrig, H.: Termination Criteria for DPO Transformations with Injective Matches. Electr. Notes Theor. Comput. Sci. 175(4) (2007)
17. Lindqvist, J., Lundkvist, T., Porres, I.: A Query Language With the Star Operator. In: Workshop on Graph Transformation and Visual Modeling Techniques (2007)
18. Minas, M., Hoffmann, B.: An Example of Cloning Graph Transformation Rules for Programming. Electr. Notes Theor. Comput. Sci. 211, 241–250 (2008)
19. Object Management Group. UML 2.0 Superstructure Specification, OMG Adopted Specification ptc/03-08-02 (August 2003)
20. Object Management Group. Business Process Modeling Notation (BPMN) Version 1.0 (May 2004)
21. Plump, D.: Confluence of Graph Transformation Revisited. In: Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday. LNCS, pp. 280–308. Springer, Heidelberg (2005)
22. Rensink, A.: Nested Quantification in Graph Transformation Rules. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 1–13. Springer, Heidelberg (2006)
23. Schürr, A., Winter, A.J., Zündorf, A.: Graph Grammar Engineering with PROGRES. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, Springer, Heidelberg (1995)
24. Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.-M.: Refactoring UML Models. In: The Unified Modeling Language, Modeling Languages, Concepts and Tools. LNCS. Springer, Heidelberg (2001)
25. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
26. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: A Unified Approach for Composing UML Aspect Models based on Graph Transformation. Transactions on AOSD - Special Issue on Aspects and Model-Driven Engineering (2008) (in press)

Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions

Esther Guerra[1], Juan de Lara[2], and Fernando Orejas[3]

[1] Universidad Carlos III de Madrid, Spain
eguerra@inf.uc3m.es
[2] Universidad Autónoma de Madrid, Spain
jdelara@uam.es
[3] Universitat Politècnica de Catalunya, Spain
orejas@lsi.upc.edu

Abstract. Pattern-based model-to-model transformation is a new approach for specifying transformations in a declarative, relational and formal style. The language relies on patterns describing allowed or forbidden relations between two models, which are compiled into operational mechanisms to perform forward and backward transformations.

In this paper, we extend the approach for handling attribute conditions expressed in some suitable logic, adapt the operational mechanisms based on graph transformation to relax attribute handling by constraint solving, and discuss heuristics for the compilation of patterns into rules.

1 Introduction

Model-to-Model (M2M) transformations are widely used in Model-Driven Engineering, e.g. to migrate between language versions, to transform into a verification domain, or to refine a model. There are two main approaches to M2M transformation: *operational* and *declarative*. The former is based on operations that explicitly state how and when creating target elements from source ones. Instead, declarative approaches describe mappings between source and target models in a direction-neutral way. Then, operational mechanisms are generated for different scenarios, e.g. to transform a source model into a target one or vice versa (forward and backward transformations), to synchronize two models, or to signal inconsistencies between them [8].

In previous work [3] we proposed a declarative, relational and formal approach to M2M transformation based on *triple patterns* to express the relations between two models. Our patterns are similar to graph constraints [6] but for triple graphs made of two graphs plus their traceability relations. Patterns can specify positive information (the relation they declare must hold) or negative one (the relation must not hold). A pattern specification is compiled into operational mechanisms, implemented with Triple Graph Grammar (TGG) operational rules [5,8,14], to perform forward and backward transformations.

In this paper we extend our framework with attributes. Traditionally, attribute handling has been one of the main difficulties of declarative bidirectional languages. For example, attribute computations must be specified in a non-causal way, and therefore generating operational mechanisms involves their algebraic manipulation for the synthesis of attribute pre-conditions and computations, which may be difficult to automate. We tackle these issues by the uniform integration of attribute computations and conditions in patterns, and by considering the manipulated models also as constraints, hence avoiding algebraic manipulation. Thus, during the transformation, attributes in models are specified by variables and formulae constraining them. When the transformation finishes, one can resort to an equation solver to obtain concrete attribute values.

The advantages of our proposal are the following. First, its relational style contrasts with declarative approaches such as TGGs, where a causality between the existing elements in the models and the ones to be created has to be given. Second, the order of pattern enforcement is deduced, contrary to approaches such as QVT, where it must be explicitly specified. Third, its formal foundation allows studying the specification in both declarative (patterns) and operational (derived rules) formats. Fourth, our patterns have a compositional style – a triple graph satisfies two patterns in conjunction if it satisfies the two patterns separately – making specifications extensible. Finally, the separation of the operational mechanism from the declarative specification allows generating operational mechanisms for different purposes, as well as using different operational languages (e.g. graph grammar rules, a constraint solver, or QVT core [13]).

Paper Organization. Section 2 introduces triple graphs, our new concept of constraints, and the algebraic approach to M2M transformation. Sections 3 and 4 present our pattern-based notation and the generation of operational rules, sketching some heuristics to improve their efficacy. Section 5 shows a case study. Section 6 compares with related work, and Section 7 ends with the conclusions.

2 Algebraic Approach to Model-to-Model Transformation

This section introduces triple graphs, constraint triple graphs, and triple graph transformation. Triple graphs are based on labelled graphs (called E-graphs in [6]), which allow data in nodes and edges. An E-graph G is defined as a special kind of graph that includes an additional set of nodes D^G with the values stored in the graph, and two additional kinds of edges that are used for attribution of nodes and edges. Mappings between E-graphs (morphisms) are tuples of set morphisms – one for each set in the E-graph – such that the structure of the E-graph is preserved (for details see [6]). For the typing we use a type graph [6], similar to a meta-model, but for simplicity we omit further discussion on types.

Triple graphs are made of three graphs: source (S), target (T) and correspondence (C). Nodes in the correspondence graph relate nodes in the source and target graphs by means of two graph morphisms [5], and for technical reasons we restrict them to be unattributed (i.e. $D^C = \emptyset$). We use triple graphs to store the source and target models of a M2M transformation, as well as the transformation traces.

Definition 1 (Triple Graph and Morphism). *A triple graph $TrG = (S \overset{c_S}{\leftarrow} C \overset{c_T}{\rightarrow} T)$ is made of three E-graphs S, C and T s.t. $D^C = \emptyset$, and two graph morphisms c_S and c_T, called the source and target correspondence functions.*

A triple morphism $m = (m_S, m_C, m_T)\colon TrG^1 \rightarrow TrG^2$ is made of three E-morphisms m_X for $X = \{S, C, T\}$, s.t. $m_S \circ c_S^1 = c_S^2 \circ m_C$ and $m_T \circ c_T^1 = c_T^2 \circ m_C$, where c_S^x and c_T^x are the correspondence functions of TrG^x (for x=$\{1, 2\}$).

We use the notation $\langle S, C, T\rangle$ for a triple graph made of graphs S, C and T. Given $TrG = \langle S, C, T\rangle$, we write $TrG|_X$ for $X \in \{S, C, T\}$ to refer to a triple graph where only the X graph is present, e.g. $TrG|_S = \langle S, \emptyset, \emptyset\rangle$. Triple graphs and morphisms form the category **TrG**.

Example 1. Fig. 1 shows a triple graph relating a class diagram and a relational schema. The graph nodes are depicted as rectangles, and the data nodes in D^S and D^T as circles. We only draw the used data nodes, as they may be infinite. Graph G in Fig. 5 shows the same triple graph using the UML notation, as well as types.

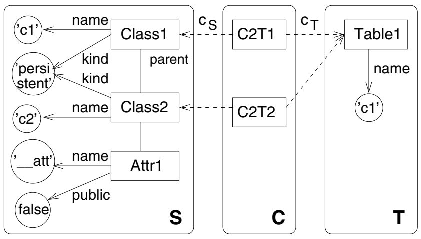


Fig. 1. Triple graph example

In order to describe the manipulation of triple graphs by means of graph transformation rules, these rules may need to include graphs storing variables that will typically be instantiated when applying the rule. Moreover, we may need to express some properties about these variables. We have formalized this kind of graph using the new notion of *constraint triple graphs*. These are triple graphs attributed over a finite set of variables, and equipped with a formula on this set to constrain the possible attribute values of source and target elements.

Definition 2 (Constraint Triple Graph). *Given an algebra $\mathcal{A}$ over signature $\Sigma = (S, OP)$, a* constraint triple graph $CTrG^{\mathcal{A}} = (TrG, \nu, \alpha)$ *consists of a triple graph $TrG = \langle S, C, T\rangle$, a finite set of S-sorted variables $\nu = D^S \uplus D^T$ (with $\uplus$ denoting disjoint union) and a $\Sigma(\nu)-$formula α in conjunctive or clausal form.*

Example 2. Fig. 2 shows a constraint triple graph. We take the convention of placing in the left compartment the terms of the formula concerning only source graph attributes; in the right compartment the terms constraining only attributes in the target; and the terms constraining both in the middle. In all cases we omit the conjunctions. Note that "=" denotes equality, not assignment. Hence, in our approach there is no attribute computation, but only attribute conditions. Finally, unused attributes are omitted in the figures, and the formula of the empty constraint is equal to **true**.

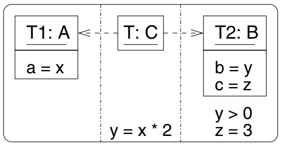


Fig. 2. Constraint

Notice that constraint triple graphs do not store data explicitly in the graphs: the data nodes D^S and D^T are variables. Thus, if for instance we would like to store a value V on an attribute node, it is enough to label that node with some fresh variable X and include the equality $X = V$ in the associated formula.

Before defining morphisms between constraints, we need an auxiliary operation for restricting $\Sigma(\nu)-$formulae to a smaller set of variables $\nu' \subseteq \nu$. This will be used when restricting a constraint triple graph to the source or target graph only (e.g. when checking the forward or backward satisfaction of the constraint). Thus, given a $\Sigma(\nu)$-formula α, its restriction to $\nu' \subseteq \nu$ is given by $\alpha|_{\nu'} = \alpha'$, where α' is like α, but with all clauses with variables in $\nu - \nu'$ replaced by **true**. For example $(x = 3 \wedge y = 7)|_{\{x\}} = (x = 3 \wedge \textbf{true}) = (x = 3)$.

Given a constraint $CTrG^{\mathcal{A}} = (TrG, \nu, \alpha)$, we write α^S for the restriction to the source variables $\alpha|_{D^S}$, and α^T for the restriction to the target variables $\alpha|_{D^T}$. Given a variable assignment $\mathfrak{f}\colon \nu \to \mathcal{A}$, we write $\mathcal{A} \models_{\mathfrak{f}} \alpha$ to denote that the algebra $\mathcal{A}$ satisfies the formula α with the value assignment induced by $\mathfrak{f}$.

Next, we define morphisms between constraint triple graphs. These are made of a triple graph morphism and a mapping of variables (i.e. a set morphism). In addition we require an implication from the formula of the constraint in the codomain to the one in the domain, as well as implications from the source and target restrictions of the formula in the codomain to the restrictions of the formula in the domain. This means that the formula in the domain constraint should be weaker or equivalent to the formula of the constraint in the codomain (intuitively, the codomain should be "more restricted").

Definition 3 (Constraint Triple Graph Morphism). *A constraint triple graph morphism $m = (m^{TrG}, m^{\nu})\colon CTrG_1^{\mathcal{A}} \to CTrG_2^{\mathcal{A}}$ is made of a triple morphism $m^{TrG}\colon TrG_1 \to TrG_2$ and a mapping $m^{\nu}\colon \nu_1 \to \nu_2$ s.t. the diagram to the left of Fig. 3 commutes, and $\forall \mathfrak{f}\colon \nu_2 \to \mathcal{A}$ s.t. $\mathcal{A} \models_{\mathfrak{f}} \alpha_2$, then $\mathcal{A} \models_{\mathfrak{f}} (\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)) \wedge (\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T)) \wedge (\alpha_2 \Rightarrow m^{\nu}(\alpha_1))$, where $m^{\nu}(\alpha)$ denotes the formula obtained by replacing every variable X in α by the variable $m^{\nu}(X)$.*

Remark 1. Note that $\alpha_2 \Rightarrow m^{\nu}(\alpha_1)$ does not imply $\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)$ or $\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T)$. For technical reasons we require $(\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)) \wedge (\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T))$ as will be evident in Definition 4 and its associated remark.

Example 3. The right of Fig. 3 shows a constraint triple graph morphism. Concerning the formula, assume some variable assignment $\mathfrak{f}\colon \nu_B \to \mathcal{A}$ satisfying α_B (i.e. $\mathcal{A} \models_{\mathfrak{f}} \alpha_B$), then such $\mathfrak{f}$ makes $\mathcal{A} \models_{\mathfrak{f}} [(x_0 = 4 \wedge z > x_0) \Rightarrow (x_0 > 0)] \wedge [(y_0 >= 1) \Rightarrow (y_0 > 0)] \wedge [(x_0 = 4 \wedge z > x_0 \wedge x_0 > y_0 \wedge w > x_0 \wedge y_0 >= 1) \Rightarrow (x_0 > 0 \wedge y_0 <> x_0 \wedge y_0 > 0)]$. In this case, the formula in the constraint A is weaker than the formula in B.

$$\begin{tikzcd}
D_S^{TrG_1} \arrow[r, "m_{D,S}^{TrG}"] \arrow[d, hook] \arrow[dr, phantom, "="] & D_S^{TrG_2} \arrow[d, hook] \\
\nu_1 \arrow[r, "m^{\nu}"] & \nu_2 \\
D_T^{TrG_2} \arrow[r, "m_{D,T}^{TrG}"] \arrow[u, hook] \arrow[ur, phantom, "="] & D_T^{TrG_2} \arrow[u, hook]
\end{tikzcd}$$

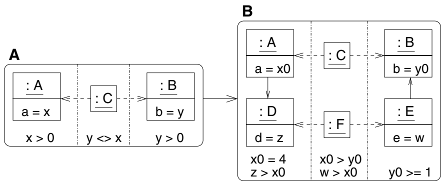


Fig. 3. Condition for **CTrG**-morphisms (left). Example (right).

From now on, we restrict to injective morphisms for the sake of simplicity, and because our patterns are made of injective morphisms. Given Σ and $\mathcal{A}$, constraint triple graphs and morphisms form the category $\mathbf{CTrG}_{\mathcal{A}}$. As we will show later, we need to manipulate objects in this category through pushouts and restrictions. A pushout is the result from gluing two objects B and C along a common subobject A, written $B +_A C$. Pushouts in $\mathbf{CTrG}_{\mathcal{A}}$ are built by making the pushout of the triple graphs, and taking the conjunction of their formulae.

Proposition 1 (Pushout in $\mathbf{CTrG}_{\mathcal{A}}$). *Given the span of* $\mathbf{CTrG}_{\mathcal{A}}$*-morphisms* $B^{\mathcal{A}} \xleftarrow{b} A^{\mathcal{A}} \xrightarrow{c} C^{\mathcal{A}}$, *its pushout is given by* $D^{\mathcal{A}} = (B +_A C, \nu_B +_{\nu_A} \nu_C, c'(\alpha_B) \wedge b'(\alpha_C))$, *and morphisms* $c' \colon B^{\mathcal{A}} \to D^{\mathcal{A}}$ *and* $b' \colon C^{\mathcal{A}} \to D^{\mathcal{A}}$ *induced by the pushouts in triple graphs* $(B +_A C)$ *and sets* $(\nu_B +_{\nu_A} \nu_C)$.

Example 4. Fig. 4 shows a pushout, where $\alpha_D \Rightarrow c'(b(\alpha_A)) \equiv b'(c(\alpha_A))$.

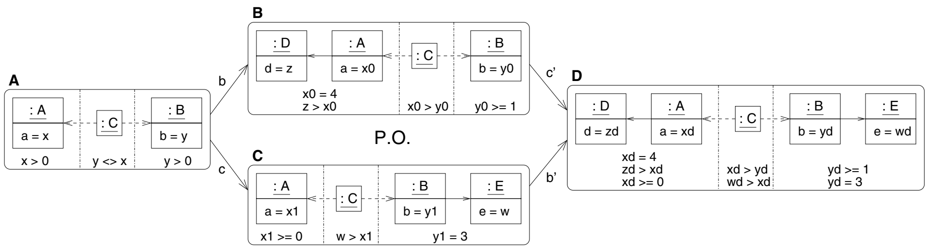


Fig. 4. Pushout example

The source restriction of a constraint triple graph is made of the source graph and the source formula, and similarly for target. This will be used later to keep just the source or target model in a constraint, when such constraint is evaluated either source-to-target or target-to-source.

Definition 4 (Constraint Restriction). *Given* $CTrG^{\mathcal{A}} = (TrG, \nu, \alpha)$, *its source restriction is given by* $CTrG^{\mathcal{A}}|_S = (TrG|_S = \langle S, \emptyset, \emptyset \rangle, D^S, \alpha|_{D^S} = \alpha^S)$. *The target restriction* $CTrG^{\mathcal{A}}|_T$ *is calculated in an analogous way.*

Remark 2. The source restriction $CTrG^{\mathcal{A}}|_S$ of a constraint induces a morphism $CTrG^{\mathcal{A}}|_S \hookrightarrow CTrG^{\mathcal{A}}$. Also, given a morphism $q \colon C^{\mathcal{A}} \to Q^{\mathcal{A}}$, we can construct morphisms $q_S \colon C^{\mathcal{A}}|_S \to Q^{\mathcal{A}}|_S$ and $q_T \colon C^{\mathcal{A}}|_T \to Q^{\mathcal{A}}|_T$.

An attributed triple graph can be seen as a constraint triple graph whose formula is satisfied by a unique variable assignment, i.e. $\exists_1 \mathfrak{f} : \nu \to \mathcal{A}$ with $\mathcal{A} \models_{\mathfrak{f}} \alpha$. We call such constraints *ground*, and they form the $\mathbf{GroundCTrG}_{\mathcal{A}}$ full subcategory of $\mathbf{CTrG}_{\mathcal{A}}$. We usually depict ground constraints with the attribute values induced by the formula in the attribute compartments and omit the formula (e.g. see constraint *CTrG* to the right of Fig. 7). The equivalence between ground constraints and triple graphs is useful as, from now on, we just need to work with constraint triple graphs. In particular, triple graphs are manipulated with TGG operational rules, but seeing them as ground constraint graphs, which offers several benefits, as we will see. The rules that we consider in this paper are non-deleting and consist of left and right hand sides (LHS and RHS) made of a constraint triple graph each, plus sets of negative pre- and post-conditions. A rule can be applied to a *host* triple graph if a constraint morphism exists from the LHS to the graph and no negative pre-condition (also called NAC) is found. Then, the rule is applied by making a pushout of the RHS and the host graph through their intersection LHS, which adds the elements created by the rule to the host graph. This step is called direct derivation. Negative post-conditions are checked after rule application, and such application is undone if they are found.

The most usual way [6, 14] of dealing explicitly with triple graphs instead of with ground constraint graphs poses some difficulties, most notably concerning attribute handling. For instance, Fig. 5 shows an example where a TGG operational rule is applied to a triple graph G. The rule creates a column for each private attribute starting by '__'. Function `LTRIM(p1,p2)` returns `p2` after removing `p1` from its beginning.

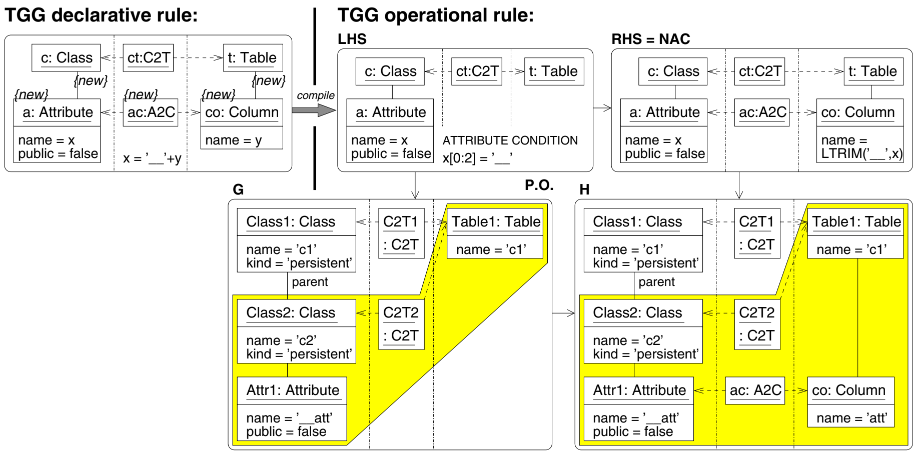


Fig. 5. Direct derivation by a non-deleting TGG operational rule

In practice, the TGG operational rules are not specified by hand, but derived from declarative rules modelling the synchronized evolution of two models [14], as depicted in the upper part of Fig. 5. The declarative rule is shown with its LHS and RHS together, and *new* tags indicating the synchronously created elements. Of course, in declarative rules, attribute computations must be expressed in a declarative style. However, their compilation into operational rules has to assign a causality to attribute computations, which involves algebraic manipulation of formulae. Moreover, appropriate attribute conditions must be synthesized too. In the example, the condition *x='__'+y* has to be transformed into a computation *LTRIM('__',x)* for the created column name, and into the condition *x[0:2]='__'* as the attribute name should start by '__'. Please note that this kind of manipulation is difficult to automate, since it involves the synthesis of operations and conditions. Our approach proposes a more straightforward solution. Fig. 8 shows the same example when dealing with triple graphs as ground constraints, where there is no need to synthesize attribute computations. The result of a transformation is a pair of models where their attributes are variables with values given by formulae. If needed, a constraint solver can compute concrete values.

3 Pattern-Based Model-to-Model Transformation

Triple Patterns are similar to graph constraints [6], but made of constraint triple graphs instead of graphs. We use them to describe the allowed and forbidden relations between source and target models in a M2M transformation.

Definition 5 (Triple Pattern). *Given the injective* $\mathbf{CTrG}_{\mathcal{A}}$*-morphism* $C \xrightarrow{q} Q$ *and the sets of injective* $\mathbf{CTrG}_{\mathcal{A}}$*-morphisms* $N_{Pre} = \{Q \xrightarrow{c_i} C_i\}_{i \in Pre}$, $N_{Post} = \{Q \xrightarrow{c_j} C_j\}_{j \in Post}$ *of negative pre- and post-conditions:*

- $\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \wedge \bigwedge_{j \in Post} \overrightarrow{N}(C_j)$ *is a positive pattern (P-pattern).*
- $\overrightarrow{N}(C_j)$ *is a negative pattern (N-pattern).*

Remark 3. The notation $\overleftarrow{P}(\cdot)$, $\overleftarrow{N}(\cdot)$, $\overrightarrow{N}(\cdot)$ and $P(\cdot)$ is just syntactic sugar to indicate a positive pre-condition (that we call parameter), a negative pre-condition, a negative post-condition and the main constraint respectively.

The simplest P-pattern is made of a main constraint Q restricted by negative pre- and post-conditions (*Pre* and *Post* sets). In this case, Q has to be present in a triple graph (i.e. in a ground constraint) whenever no negative pre-condition C_i is found; and if Q is present, no negative post-condition C_j can be found. While pre-conditions express restrictions for the constraint Q to occur, post-conditions describe forbidden graphs. If a negative pre-condition is found, it is not mandatory to find Q, but still possible. P-patterns can also have parameters, specified with a non-empty C. In such case, Q has to be found only if C is also found. Finally, an N-pattern is made of one negative post-condition, forbidden to occur (and hence C and Q are empty).

Example 5. The left of Fig. 6 shows a P-pattern, taken from the class to relational transformation [13]. It is made of a main constraint `C-T` with a negative pre-condition `Parent`. It maps persistent classes without parents to tables with the same name. The negative pre-condition shows only the elements that do not belong to the main constraint, and those connected to them.

The right of Fig. 6 shows a P-pattern with its parameters indicated with ⟨⟨`param`⟩⟩. We present C and Q together, as usually the formula in C is the same as the one in Q. The pattern maps the attributes of a class with the columns of the table related to the class. As Section 4.1 will show, it is not even necessary to specify the parameters, as our heuristics are able to suggest them. In fact, a

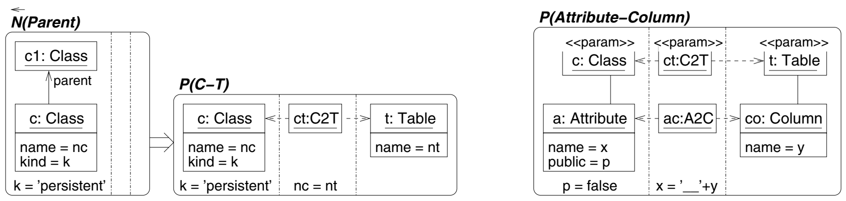


Fig. 6. P-pattern examples

M2M specification is usually made of N- and P-patterns without parameters. For technical reasons, we assume that no P-pattern has negative post-conditions.

Definition 6 (M2M Specification). *A M2M specification $SP = \bigwedge_{i\in I} P_i$ is a conjunction of patterns, where each P_i can be positive or negative.*

Next we define pattern satisfaction. A unique definition is enough as N-patterns are a special case of P-patterns. We check satisfiability of patterns on constraint triple graphs, not necessarily ground. This is so because, during a transformation, the source and target models do not need to be ground. When the transformation finishes we can use a solver in order to find an attribute assignment satisfying the formulae.

We define forward and backward satisfaction. In the former we check that the main constraint of the pattern is found in all places where the pattern is source-enabled (roughly, in all places where the pre-conditions for enforcing the pattern in a forward transformation hold). The separation between forward and backward satisfaction is useful because, e.g. if we transform forwards (assuming an initial empty target) we just need to check forward satisfaction. Full satisfaction implies both forward and backward satisfaction and is useful to check if two graphs are actually synchronized.

Definition 7 (Satisfaction). *A constraint triple graph $CTrG$ satisfies $CP = [\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \wedge \bigwedge_{j\in Post} \overrightarrow{N}(C_j)]$, written $CTrG \models CP$, iff:*

– *CP is* forward satisfiable*, $CTrG \models_F CP$: $[\forall m^S : P_S \to CTrG$ s.t. $(\forall i \in Pre$ s.t. $N_i^S \not\cong P_S$, $\nexists n_i^S : N_i^S \to CTrG$ with $m^S = n_i^S \circ a_i^S)$, $\exists m : Q \to CTrG$ with $m \circ q^S = m^S$, s.t. $\forall j \in Post\ \nexists n_j : C_j \to CTrG$ with $m = n_j \circ c_j]$, and*
– *CP is* backward satisfiable*, $CTrG \models_B CP$: $[\forall m^T : P_T \to CTrG$ s.t. $(\forall i \in Pre$ s.t. $N_i^T \not\cong P_T, \nexists n_i^T : N_i^T \to CTrG$ with $m^T = n_i^T \circ a_i^T)$, $\exists m : Q \to CTrG$ with $m \circ q^T = m^T$, s.t. $\forall j \in Post\ \nexists n_j : C_j \to CTrG$ with $m = n_j \circ c_j]$,*

with $P_x = C +_{C|_x} Q|_x$, $N_i^x = C +_{C|_x} C_i|_x$ and $N_i^x \xleftarrow{a_i^x} P_x \xrightarrow{q^x} Q$ $(x \in \{S, T\})$, see left of Fig. 7. $C +_{C|_x} Q|_x$ is the pushout object of C and $Q|_x$ through $C|_x$.

In forward satisfaction, for each occurrence of $P_S = C +_{C|_S} Q|_S$ satisfying the negative pre-conditions, an occurrence of Q must be found satisfying the negative post-conditions. A pattern is satisfied either because no occurrence of P_S

$$\begin{tikzcd}
C_i|_S \arrow[dd, "d_i^S"'] \arrow[dr, phantom, "P.O."] & & C|_S \arrow[ll, "b_S^i"'] \arrow[dl, bend right] \arrow[dr, "q_S"] \arrow[d, phantom, "P.O."] & & \\
& C \arrow[dl, "e_i^S"'] \arrow[dr, "c^S"] \arrow[d, phantom, "="] & & Q|_S \arrow[dl, "p^S"] & \\
N_i^S \arrow[drr, bend right, "n_i^S"', "/" marking] & = & P_S \arrow[ll, "a_i^S"] \arrow[r, "q^S"] \arrow[d, "m^S"'] & Q \arrow[r, "c_j"] \arrow[dl, "m"] & C_j \arrow[dll, bend left, "n_j", "/" marking] \\
& & CTrG & &
\end{tikzcd}$$

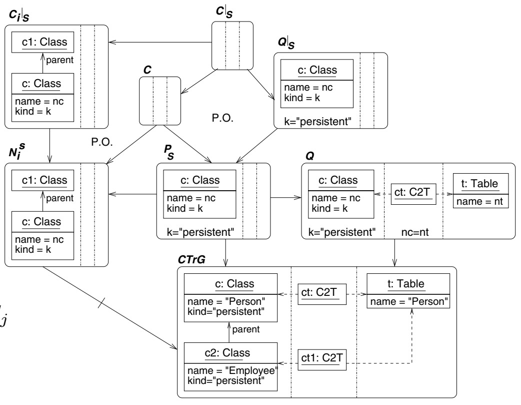


Fig. 7. Forward satisfaction (left). Example (right).

exists (*trivial satisfaction*), because some occurrence of P_S exists as well as some occurrence of the negative pre-conditions (*vacuous satisfaction*), or because an occurrence of the main constraint Q exists, and none of the negative pre- and post-conditions (*positive satisfaction*). Note that if the resulting negative pre-condition N_i^x is isomorphic to P_x, it is not taken into account. This is needed as many pre-conditions express a restriction in either source or target but not on both. Similar conditions are demanded for backward satisfiability.

Example 6. The right of Fig. 7 depicts the satisfaction of pattern `C-T` shown in Fig. 6 by the ground constraint $CTrG$. We have $CTrG \models_F C - T$ as there are two occurrences of P_S, and the first one (shown by equality of identifiers) is positively satisfied, while the second (node $c2$) is vacuously satisfied. We also have $CTrG \models_B C - T$, as there is just one occurrence of P_T, positively satisfied. Hence $CTrG \models C - T$.

Given a specification $SP = \bigwedge_{i \in I} P_i$ and a constraint $CTrG$, we write $CTrG \models SP$ to denote that $CTrG$ satisfies all patterns in SP. The semantics of a specification is the language of all constraint triple graphs that satisfy it.

Definition 8 (Specification Semantics). *Given a specification SP, its semantics is given by $SEM(SP) = \{CTrG \in Obj(\mathbf{CTrG}_{\mathcal{A}}) | CTrG \models SP\}$, where $Obj(\mathbf{CTrG}_{\mathcal{A}})$ are all objects in the category $\mathbf{CTrG}_{\mathcal{A}}$.*

The semantics is defined as a set of constraint triple graphs, not necessarily ground. Given a non-ground constraint, a solver can obtain a ground constraint satisfying it, if it exists. Moreover, the specification semantics is compositional, as adding new patterns to a specification amounts to intersecting the languages of both. This fact is useful when extending or reusing pattern-based specifications.

Proposition 2 (Composition of Specifications). *Given specifications SP_1 and SP_2, $SEM(SP_1 \wedge SP_2) = SEM(SP_1) \cap SEM(SP_2)$.*

4 Generation of Operational Mechanisms

This section describes the synthesis of TGG operational rules implementing forward and backward transformations from pattern-based specifications. In forward transformation, we start with a constraint triple graph with correspondence and target empty, and the other way round for backward transformation.

The synthesis process creates from each P-pattern one rule that contains triple constraints in its LHS and RHS. In particular, $P_S = C +_{C|_S} Q|_S$ is taken as the LHS for the forward rule and Q as the RHS. The negative pre- and post-conditions of the P-pattern are used as negative pre- and post-conditions of the rule. All N-patterns are converted into negative post-conditions of the rule, using the well-known procedure to convert graph constraints into rule's post-conditions [6]. Finally, additional NACs are added to ensure termination.

Definition 9 (Derived Operational Rules). *Given specification SP and $P = [\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \wedge \bigwedge_{j \in Post} \overrightarrow{N}(C_j)] \in SP$, the following rules are derived:*

- **Forward.** $\overrightarrow{r_P} : ((L = C +_{C|_S} Q|_S \rightarrow R = Q), pre^S(P), post(P)),$
- **Backward.** $\overleftarrow{r_P} : ((L = C +_{C|_T} Q|_T \rightarrow R = Q), pre^T(P), post(P)),$

where $pre^x(P)$ (for $x = \{S, T\}$) is the union of the following two sets of NACs:

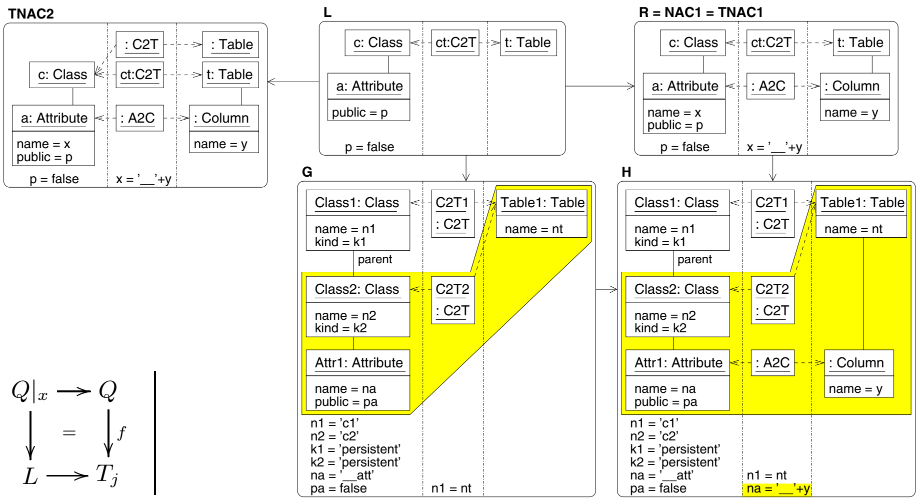
- *$NAC^x(P) = \{L \xrightarrow{a_i^x} N_i^x | L \ncong N_i^x\}_{i \in Pre}$ is the set of NACs derived from P's negative pre-conditions, with $N_i^x \cong C_i|_x +_{C|_x} C$.*
- *$TNAC^x(P) = \{L \xrightarrow{n_j} T_j\}$ is the set of NACs ensuring termination, where T_j is built by making n_j injective and jointly surjective with $Q \xrightarrow{f} T_j$, s.t. the diagram to the bottom-left of Fig. 8 commutes.*

and $post(P)$ is the union of the following two sets of negative post-conditions:

- *$POST(P) = \{n_j : R \rightarrow C_j\}_{j \in Post}$ is the set of rule's negative post-conditions, derived from the set of P's post-conditions.*
- *$NPAT(P) = \{R \rightarrow D | [\overrightarrow{N}(C_k)] \in SP$, $R \rightarrow D \leftarrow C_k$ is jointly surjective, and $(R \setminus L) \cap C_k \neq \emptyset\}$ is the set of negative post-conditions derived from each N-pattern $\overrightarrow{N}(C_k) \in SP$.*

The set $NPAT(P)$ contains the negative post-conditions derived from the N-patterns of the specification. This is done by merging each N-pattern with the rule's RHS in all possible ways. Moreover, the condition $(R \setminus L) \cap C_k \neq \emptyset$ reduces the size of $NPAT(P)$, by only considering violations of the N-patterns due to the creation of elements, as we start with an empty target model.

Example 7. The upper row of Fig. 8 shows the operational forward rule generated from pattern `Attribute-Column`. The set NAC^S contains one constraint, equal to R. There are two NACs for termination, `TNAC2` and `TNAC1`, the latter



$$\begin{tikzcd} Q|_x \arrow[r] \arrow[d] \arrow[dr, phantom, "="] & Q \arrow[d, "f"] \\ L \arrow[r] & T_j \end{tikzcd}$$

Fig. 8. Condition for $TNAC^x(P)$ (left). Example rule and derivation (right).

equal to R. As a difference from Fig. 5, we do not need to do algebraic manipulation of formulae to generate the rule. The figure also shows a direct derivation where both G and H are ground constraints. Note also that we do not check in L that x starts with "__", but if it does not, we would obtain an unsatisfiable constraint.

According to [12], the generated rules are terminating, and in absence of N-patterns, correct: they produce only valid models of the specification. However, the rules are not complete: not all models satisfying the specification can be produced. Next subsection describes a method, called *parameterization*, that in addition ensures completeness of the rules generated from a specification without N-patterns. If a specification contains N-patterns, these are added as negative post-conditions for the rules, preventing the occurrence of N-patterns in the model. However, they may forbid applying any rule before a valid model is found, thus producing graphs that may not satisfy all P-patterns (because the transformation stopped too soon). That is, in this situation the operational mechanism would not be able to find a model, even if it exists. Next subsection presents one heuristic that ensures finding models, and hence correctness, for mechanisms derived from some specifications with certain classes of N-patterns.

4.1 Parameterization and Heuristics for Rule Derivation

Applying the *parameterization* operation to each P-pattern in the specification ensures completeness of the operational mechanism: the rules are able to generate all possible models of the specification [12]. The operation takes a P-pattern and generates additional ones, with all possible positive pre-conditions "bigger" than the original pre-condition, and "smaller" than the main constraint Q. This allows the rules generated from the patterns to reuse already created elements.

Definition 10 (Parameterization). *Given* $T = \bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \wedge \bigwedge_{j \in Post} \overrightarrow{N}(C_j)$*, its parameterization is* $Par(T) = \{ \bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C') \Rightarrow P(Q) \wedge \bigwedge_{j \in Post} \overrightarrow{N}(C_j) | C \overset{i_1}{\hookrightarrow} C' \overset{i_2}{\hookrightarrow} Q, C \ncong C', C' \ncong Q \}$.

Remark 4. The formula $\alpha_{C'}$ can be taken as the conjunction of α_C for the variables already present in ν_C, and α_Q for the variables not in ν_C (i.e. in $\nu'_C \setminus i_1(\nu_C)$). Formally, $\alpha_{C'} = \alpha_C \wedge \alpha_Q|_{i_2(\nu_{C'} \setminus i_1(\nu_C))}$ (assuming no renaming of variables).

Example 8. Fig. 9 shows an example, where some of the parameters generated by parameterization are shown for a pattern like the one in Fig. 6 but without parameters. Parameterization generates 45 patterns, each one made of the same main constraint and one of the generated parameters. The pattern with parameter $\overrightarrow{P}(1)$ is enforced when the class is already mapped to a table, and in forward transformation avoids generating a rule that creates a table with arbitrary name. Parameter $\overrightarrow{P}(3)$ reuses a column with the same name as the attribute (up to the preffix '__'), possibly created by a parent class. However, $\overrightarrow{P}(2)$ is harmful as it may lead to reusing a column connected to a different table, and thus to an incorrect model where the column is connected to two different tables.

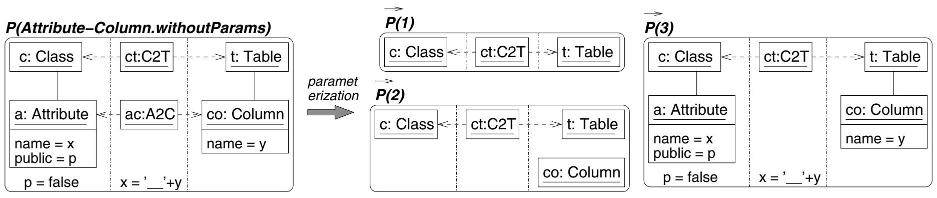


Fig. 9. Parameterization example

As the example shows, parameterization generates an exponential number of patterns with increasingly bigger parameters, which may lead to operational rules reusing too much information. Although this ensures completeness, we hardly use it in practice, and we prefer using heuristics to control the level of reuse. However, as previously stated, generating fewer patterns can make the rules unable to find certain models of the specifications (those "too small").

We propose two heuristics in this paper. The first one is used to derive only those parameters that avoid creation of elements with unconstrained attribute values. The objective is to avoid synthesizing rules creating elements whose attributes can take several values.

Heuristic 1. *A pattern P can be replaced by another one having the same main constraint and as parameter all elements with attributes not constrained by any formula, and the mappings between these elements.*

Example 9. In Fig. 9, the heuristic generates just one pattern with parameter $\overrightarrow{P}(1)$. Thus, the generated rules avoid creating a table with arbitrary name.

Next heuristic generates only those parameters that avoid duplicating a graph S_1, forbidden by some N-pattern of the form $\overrightarrow{N}(S_1 +_U S_1)$, preventing the subgraph S_1 to appear twice. This ensures the generation of rules producing valid models for specifications with N-patterns of this form (called FIP in [3]).

Heuristic 2. *Given* $[\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q)]$, *if* $[\overrightarrow{N}(S)] \in SP$ *with* $S \cong S_1 +_U S_1$, *and* $\exists s\colon S_1 \to Q$ *and* $\nexists s'\colon S_1 \to C$ *both injective s.t.* $q \circ s' = s$, *we generate additional patterns with parameters all* C'_j *s.t.* q_1 *and* q_s *in* $C \xrightarrow{q_1} C'_j \xleftarrow{q_s} S_1$ *are jointly surjective, and the induced* $C'_j \to Q$ *is injective.*

The way to proceed is to apply heuristic 2 to each P- and N-pattern of the form $\overrightarrow{N}(S_1 +_U S_1)$, and repeat the procedure with the resulting patterns until no more different patterns are generated. Next section illustrates both heuristics.

5 Example

Next we illustrate our approach with a bidirectional transformation between relational database schemas (RDBMS) and XML documents. Their meta-models are shown in the meta-model triple in Fig. 10. Schemas contain books and subjects. A book has zero or more subjects, and those books with the same subject description are related to the same object *Subject*. On the contrary, the XML meta-model allows nested relationships, and even if two books have the same subject description, they are assigned two different objects *Subject*.

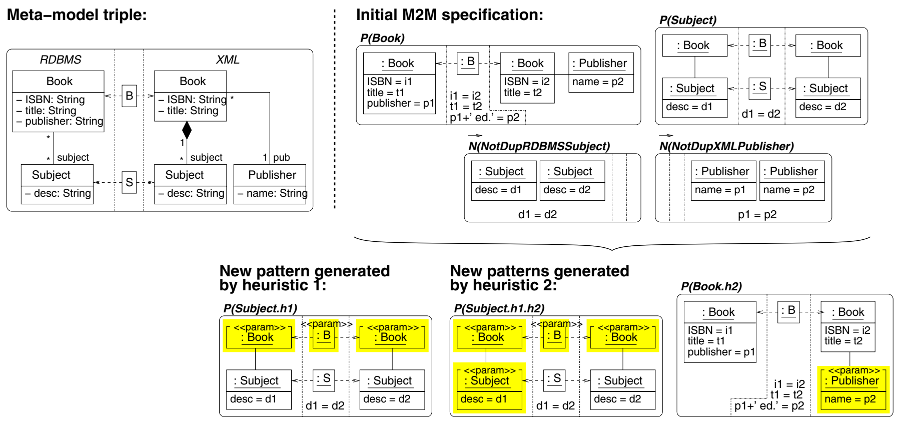


Fig. 10. Mapping relational database schemas and XML

Fig. 10 shows the initial M2M specification, which is made of four patterns. The P-pattern `Book` states how the books in both meta-models should relate, and adds an " ed." suffix to the publisher in the XML model. P-pattern `Subject` maps subjects in both models. Note that we need these two patterns as it is possible to have books with zero or more subjects. Should a book have exactly one subject, then only one pattern would have been enough. In addition, as the RDBMS format does not allow two subjects with the same description, we forbid such situation by defining the N-pattern `NotDupRDBMSSubject`. Similarly, N-pattern `NotDupXMLPublisher` forbids repeating publishers in XML.

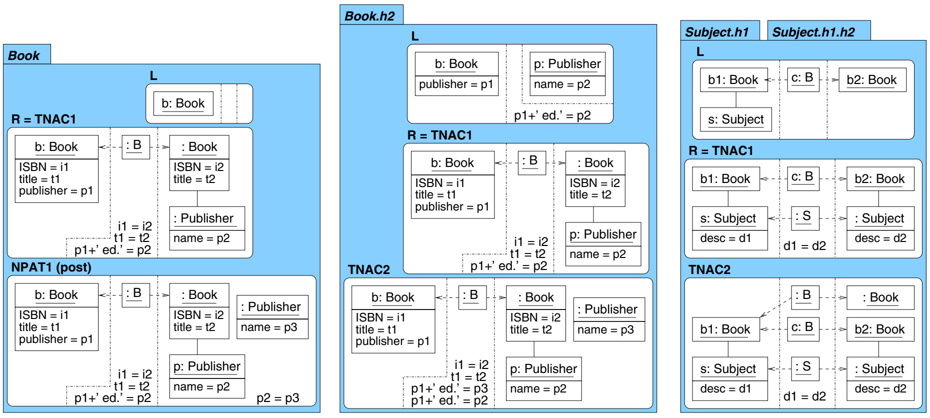


Fig. 11. Generated forward rules

In this example we cannot use generic parameterization as it would generate patterns with parameters reusing, e.g. the *Subjects* in the XML model. Therefore we use the heuristics instead. The first one generates pattern `Subject.h1` from pattern `Subject` by defining the elements with unconstrained attributes as parameters. The new pattern replaces the old one and ensures that, when the subject is translated, the book associated to it has been translated first. The second heuristic is applied to patterns `Subject.h1` and `Book` and produces two new patterns, `Subject.h1.h2` and `Book.h2`. The first one reuses RDBMS *Subjects* so that they are not duplicated in backward transformations. The second reuses one *Publisher*, avoiding its duplication in forward transformations.

As a last step, we use patterns `Book`, `Subject.h1`, `Subject.h1.h2` and `Book.h2` and the N-patterns to generate the operational rules. Fig. 11 shows the forward ones. Rule `Book` contains a termination NAC (`TNAC1`) equal to its RHS and a negative post-condition (generated from $\overrightarrow{N}$(`NotDupXMLPublisher`)) avoiding two publishers with same name. Patterns `Subject.h1` and `Subject.h1.h2` produce equivalent rules with two termination NACs. Finally, rule `Book.h2` creates books that reuse publishers once they have been created. Note again that we do not need to perform algebraic manipulation of expressions for rule synthesis, as the LHSs and RHSs contain constraint triple graphs (where note that attributes not used in formulae are ommited, like in the LHS of rule *Book*).

Altogether, the operational mechanisms generated for this example are terminating, confluent, correct and complete even using heuristics. However, our mechanisms cannot guarantee confluence in general if we do not have a means to prefer one resulting model or another.

6 Related Work

Some declarative approaches to M2M transformation use a textual syntax, e.g. PMT [15], Tefkat [9]. All of them are uni-directional, whereas our patterns are bidirectional. There are also bidirectional textual languages, like MTF [10].

Among the visual declarative approaches, a prominent example is QVT-relational [13]. Relations in QVT may include *when* and *where* clauses that identify pre- and post-conditions and can refer to other relations. From this specification, executable QVT-core is generated. This approach is similar to ours, but we compile patterns to TGG rules, allowing its analysis [6]. Besides, we can analyse the patterns themselves. In the QVT-relations language, there is no equivalent to our N-patterns. Notice however, that our N-patterns can be used to model *keys* in QVT (e.g. elements that should have a unique identifier) as we showed in Section 5 with N-patterns $\overrightarrow{N}$(`NotDupXMLPublisher`) and $\overrightarrow{N}$(`NotDupRDBMSSubject`). An attempt to formalize QVT-core is found in [7].

In [1], transformations are expressed through positive declarative relations, heavily relying on OCL constraints, but no operational mechanism is given to enforce such relations. In BOTL [2], the mapping rules use a UML-based notation that allows reasoning about applicability or meta-model conformance.

Declarative TGGs [14] formalize the synchronized evolution of two graphs through declarative rules from which TGG operational rules are derived. We also generate TGG operational rules, but whereas declarative TGG rules must say which elements should exist and which ones are created, our heuristics infer such information. Moreover, TGGs need a control mechanism to guide the execution of the operational rules, such as priorities [8] or their coupling to editing rules [5], while our patterns do not need it. As in QVT, there is no equivalent to our N-patterns, however TGGs can be seen as a subset of our approach, where a declarative TGG rule is a pattern of the form $\overleftarrow{P}(L) \Rightarrow P(R)$.

In [11] the authors start from a forward transformation and the corresponding backward transformation is derived. Their transformations only contain injective functions to ensure bidirectionality, and if an attribute can take several values one of them is chosen randomly. Finally, in [4] attribute grammars are used as transformation language, where the order of execution of rules is automatically calculated according to the dependencies between attributes.

7 Conclusions and Future Work

In this paper we have extended pattern-based transformation with attributes. The resulting language allows expressing relations between models in a declarative way, leaving open the kind of logic used for attribute conditions. Typically, it can be first order predicate logic, e.g. with OCL syntax. The advantage of our approach is that it provides a formal, high-level language to express bidirectional transformations. Our language is concise, as its heuristics allow omitting the parameters in the relations. Moreover, at the operational level, we have proposed a new way of triple graph rewriting based on constraints. This idea, which can be used in other transformation approaches, avoids manipulation of attribute conditions, one of the main difficulties of relational approaches.

We are currently working towards using this approach to formalize QVT relations. Also, we are considering other operational languages, further heuristics, devising analysis methods, and implementing a prototype tool.

Acknowledgments. Work supported by the Spanish Ministry of Science and Innovation, projects METEORIC (TIN2008-02081), MODUWEB (TIN2006-09678) and FORMALISM (TIN2007-66523). Moreover, part of this work was done during a sabbatical leave of the third author at TU Berlin, with financial support from the Ministerio de Ciencia e Innovación (grant ref. PR2008-0185). We thank the referees for their useful comments.

References

1. Akehurst, D.H., Kent, S.: A relational approach to defining transformations in a metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002)
2. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. ENTCS 72(3) (2003)
3. de Lara, J., Guerra, E.: Pattern-based model-to-model transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 426–441. Springer, Heidelberg (2008)
4. Dehayni, M., Féraud, L.: An approach of model transformation based on attribute grammars. In: Konstantas, D., Léonard, M., Pigneur, Y., Patel, S. (eds.) OOIS 2003. LNCS, vol. 2817, pp. 412–423. Springer, Heidelberg (2003)
5. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer, Heidelberg (2006)
7. Greenyer, J.: A study of model transformation technologies: Reconciling TGGs with QVT. Master's thesis, University of Paderborn (2006)
8. Königs, A., Schürr, A.: Tool integration with triple graph grammars - a survey. ENTCS 148(1), 113–150 (2006)
9. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
10. MTF. Model Transformation Framework, http://www.alphaworks.ibm.com/tech/mtf
11. Mu, S.-C., Hu, Z., Takeichi, M.: Bidirectionalizing tree transformation languages: A case study. JSSST Computer Software 23(2), 129–141 (2006)

12. Orejas, F., Guerra, E., de Lara, J., Ehrig, H.: Correctness, completeness and termination of pattern-based model-to-model transformation (2009) (submitted), http://astreo.ii.uam.es/~jlara/papers/compPBT.pdf
13. QVT (2005), http://www.omg.org/docs/ptc/05-11-01.pdf
14. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
15. Tratt, L.: A change propagating model transformation language. JOT 7(3), 107–126 (2008)

Towards Model Round-Trip Engineering: An Abductive Approach

Thomas Hettel[1,2], Michael Lawley[3], and Kerry Raymond[1]

[1] School of Information Technology
Queensland University of Technology, Brisbane, Australia
t.hettel@student.qut.edu.au, k.raymond@qut.edu.au
[2] SAP Research, CEC Brisbane, Australia
t.hettel@sap.com
[3] The Australian E-Health Research Centre,
CSIRO ICT Centre, Brisbane, Australia
michael.lawley@csiro.au

Abstract. Providing support for reversible transformations as a basis for round-trip engineering is a significant challenge in model transformation research. While there are a number of current approaches, they require the underlying transformation to exhibit an injective behaviour when reversing changes. This however, does not serve all practical transformations well. In this paper, we present a novel approach to round-trip engineering that does not place restrictions on the nature of the underlying transformation. Based on abductive logic programming, it allows us to compute a set of legitimate source changes that equate to a given change to the target model. Encouraging results are derived from an initial prototype that supports most concepts of the Tefkat transformation language.

1 Introduction

In the vision of model-driven software development, models are the prime artefacts. They undergo a process of gradual refinement turning high level descriptions of a system into detailed models and finally code. As part of this process, models are translated to various modelling languages most appropriately expressing important concepts of particular abstraction layers and from certain perspectives. Once models have been translated, the results are subject to revision and there is no easy way to reflect changes back to their original source. However, propagating changes is indispensable in order to keep the interconnected mesh of models in a consistent state.

While there are many different approaches to model synchronisation, they place restrictions on the underlying transformation. Generally, transformations are required to exhibit some injective behaviour such that unique source models can be found for each and every change to the target model. However, there are many practical transformations where such an injective behaviour is not achievable (e.g., see Sec. 3.1) as important information is discarded in the transformation process and not encoded in the target model. In general there are

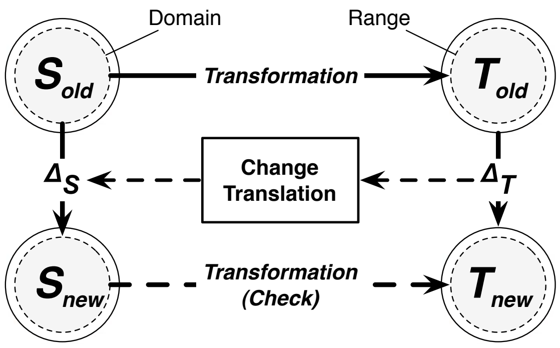


Fig. 1. Synchronisation through change translation: changes to the range in model T are translated into corresponding changes to the domain in S

often many different ways to reflect changes to the target in terms of changes to the source model and no simple decision can be made to prefer one over another.

To cope with such scenarios, this paper presents an approach to model round-trip engineering (RTE) based on unidirectional, non-injective transformations. Borrowing from abductive reasoning, a number of different source changes can be computed that all equate to the desired target change. Building upon our previous work on the formal foundations of model synchronisation [1], changes performed on the target model are translated into changes to the source. Target changes can either be relevant, manipulating the range of the transformation or irrelevant, in which case the synchronised state is not impaired and no change translation is necessary. In the following (w.l.o.g.) we only consider relevant changes. Translating changes must ensure that applying the transformation to the changed source yields exactly the range (relevant part) of new target model (Fig. 1). No other changes, called side effects, are permissible. In this paper we present an implementation of this change translation function.

The remainder of this paper is structured as follows. Sec. 2 introduces the concept of abductive reasoning and outlines how it can be leveraged to solve the round-trip engineering problem. Illustrating our approach, Sec. 3 introduces the Tefkat model transformation language alongside a running example that is used throughout this paper. The main contribution, Sec. 4, details how the idea of abduction together with other techniques can be used to effectively reverse a unidirectional transformation based on Tefkat to synchronise two models. The presented ideas are then compared to related work in Sec. 5. Concluding, we summarise and discuss our findings and provide an outlook to future work.

2 Abduction and RTE

Abduction was introduced as an epistemological theory to scientific knowledge acquisition by C.S. Pierce [2]:

"The surprising fact, C, is observed. But if A were true, C would be a matter of course. Hence, there is reason to suspect that A is true."

Arriving at A is the process of abduction also paraphrased as the "*inference to the best explanation*". To illustrate this, consider the abductive feat achieved by Johannes Kepler (1571–1630). He noticed that Mars' orbit around the sun did not comply to a circular trajectory that had been attributed to planetary motion. After years of studying planetary motion and generalising from Mars to all planets[1], he came up with his theory that planets follow an elliptical trajectory around the sun rather than a circular motion. Kepler's laws of planetary motion still hold today and correctly describe the orbits of planets and comets discovered long after Kepler's death.

Due to Pierce's broad definition of abduction, contrasting it against induction is difficult and largely depends on its concrete interpretation. Some philosophers regard abduction as a special case of induction. Others maintain it is more general and subsumes induction [3].

2.1 Abductive Logic Programming

In this paper we adopt the much narrower interpretation of abduction as pursued by the logic programming and artificial intelligence community [4,5]. The main difference is that an existing, possibly incomplete, but fixed background theory is required. By making certain assumptions, the abduction process can complete this theory so as to provide an explanation for an observed phenomenon. These incomplete parts of the theory, for which it is not known whether they hold true or not, are called *abducibles.*

Formally speaking, an abductive framework is a triple (P, A, I), where

– P is the program (or theory), a set of logic implications;
– A the set of abducibles, predicates used in P that can be assumed as required; and
– I the set of integrity constraints over predicates in A.

In order to explain an observed phenomenon Q, the abductive query, a hypothesis $H \subset A$ is sought such that:

– $H \cup P \models Q$, the hypothesis applied to the program explains the observation;
– $H \cup P \models I$, the hypothesis and the program comply with the integrity constraints; and
– $H \cup P$ *is consistent*, i.e., the hypothesis does not contradict the program.

To arrive at an explanation, the abductive logic programming (ALP) proof procedure [6] can be applied. It leverages an algorithm similar to backwards chaining, which is for instance used in Prolog implementations. An abductive explanation for Q is produced by unfolding it against the logic program or theory P. If the

[1] This generalisation is not obvious as the orbits of the other known planets are less eccentric and could crudely be approximated by a circle.

procedure encounters an abducible, it is assumed as required (i.e. such that Q succeeds) and the integrity checking phase is entered to verify that the hypothesis does not violate the constraints. While doing so, other abducibles may be encountered, assumed and checked as well. Eventually, the proof procedure will terminate with a set of hypotheses that all constitute legitimate explanations for Q with respect to the aforementioned criteria. However, it may also happen that no explanation can be found. In this case, the observation Q cannot, under no legitimate assumptions, be explained by the theory P.

2.2 Reversing and RTE as an Abductive Problem

The idea of abduction can be applied to reverse model transformations. This is achieved by interpreting the new target model as the observed phenomenon Q, which should be explained in terms of the transformation that corresponds to the program P by hypothesising about the existence of source model elements, which correspond to H and A respectively. Possible explanations are constraint by the source meta-model in terms of the defined type hierarchy and cardinality and nature of references (association vs. aggregation).

By extending the previous interpretation, also incremental RTE scenarios, which are our primary concern, can be covered as illustrated in Fig. 2. Therefore, we assume the old source and target models are accessible and only changes need to be propagated. Moreover, it is assumed that the old target model is the result of applying the transformation to the old source. In this case, the program P corresponds to the old source and target models, the trace connecting both and the transformation producing the new target from the new source. Changes to the target are represented by the observation Q and are explained in terms of source changes as part of the hypothesis H. As aforementioned, explanations have to comply with the integrity constraints I, which are mainly derived from the meta-model. This rather abstract interpretation of RTE as an abductive problem will be further refined and elaborated on in Sec. 4.

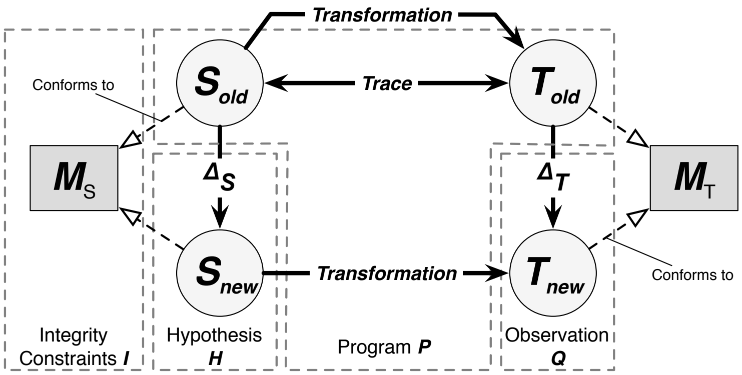


Fig. 2. Model round-trip engineering interpreted as an abductive problem. M_S and M_T represent the meta-models of source model S and target model T respectively.

3 Tefkat

To illustrate how abduction can be leveraged to facilitate model synchronisation based on a unidirectional, non-injective transformation between two models, the Tefkat transformation language [7] is used. It provides a rule-based and declarative way to specify transformations. Guaranteeing confluence of the transformation result, rules are automatically scheduled by the engine.

```
RULE   class2table
FORALL Class c
WHERE  c.persistent AND c.name = n
MAKE   Table t FROM t4c(c)
SET    t.name = append("tbl",n);

PATTERN hasAttribute(c,a)
FORALL  Class c2
WHERE   c.ownsAttr = a
OR      (c.super = c2
AND     c2.ownsAttr = a);
```

```
RULE   attr2col
FORALL Class c, Attribute a
WHERE  hasAttribute(c,a)
       AND c.persistent
       AND a.name = n
MAKE   Table t FROM t4c(c),
       Column col FROM col(c,a)
SET    t.cols = col,
       col.name = n;
```

Fig. 3. Model transformation rules given in Tefkat [7] for mapping UML class diagrams onto relational database schema

Facilitating reuse of transformation fragments, Tefkat offers rule inheritance and a concept called `PATTERN` for reusing pattern definitions. Rules consist of 3 parts (refer to Fig. 3 for examples). The source pattern (`FORALL` and `WHERE`), the target pattern (`MAKE` and `SET`) and the trace (`FROM`) connecting both. Elements that are matched as part of the source pattern can be used to uniquely identify target elements through a function[2]. Using the same function in different rules makes sure that a target object is only created once and can subsequently be reused in other rules. For instance in the example depicted in Fig. 3 the function `t4c(c)` in the `MAKE` statement of rule `class2table` creates one `Table` per `Class c`. The same function is reused in rule `attr2col` for adding `Columns` to `Tables`. In other words: `Class c` together with the function *t4c* uniquely identifies `Table t`. This mapping produced during the transformation process is stored in the trace, which can be queried for relations between source and target model.

3.1 Running Example

To illustrate the concepts introduced in the paper the following running example is used, which is based on the popular UML to relational-database-schema mapping. The transformation (see Fig. 3) is not injective as there are at least two different class diagrams that equate to the database schema depicted in Fig. 4. One is also depicted in Fig. 4, another can be derived by flattening the class hierarchy and effectively moving the *name* attribute to *Student* and *Staff*.

[2] This does not mean that Tefkat is restricted to injective transformations in any way. Rather the identity of target elements is restricted to one particular set of source elements to allow different rules to define different aspects of one target element.

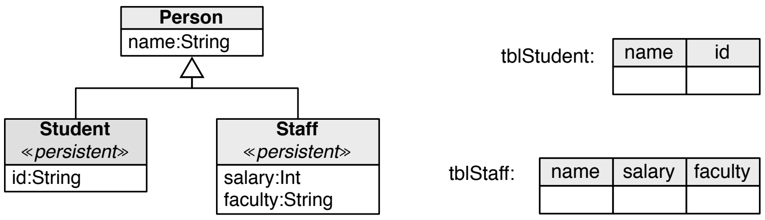


Fig. 4. A simple UML class diagram and the corresponding relational database schema with respect to the transformation depicted in Fig. 3

4 Reversing Transformations

There are a number of steps involved in reversing transformations. The following subsections elaborate on each of these steps.

4.1 Logic Programming Representation

Employing abductive logic programming to solve the RTE problem requires that models and transformation are represented in terms of first-order logic constructs. The source and target models are encoded using three different predicates for instances, attributes and references. One further predicates is needed to encode the trace:

- $inst(o, t)$, where o is a unique object identifier and t refers to a specific type;
- $attr(o, a, v)$, where o identifies the object, a the attribute and v the value;
- $ref(o_1, r, o_2)$, where o_1 is the source object and r the reference pointing to object o_2; and
- $trace(i, t, s)$ where i is the name of the uniqueness function, t the target element uniquely identified by i and the source elements s.

The following predicates encode (parts of) the UML class diagram as depicted in Fig. 4.[3] In the following, `atoms` are denoted by `type writer font`, whereas *variables* are denoted by *italics*.

$$\begin{array}{ll} inst(\texttt{person}, \texttt{class}), & ref(\texttt{person}, \texttt{attributes}, \texttt{name}), \\ attr(\texttt{person}, \texttt{name}, \text{`}\texttt{Person}\text{'}), & inst(\texttt{student}, \texttt{class}), \\ inst(\texttt{name}, \texttt{attribute}), & ref(\texttt{student}, \texttt{super}, \texttt{person}), \\ attr(\texttt{name}, \texttt{name}, \text{`}\texttt{name}\text{'}), & \ldots \end{array}$$

Transformation rules can be interpreted as logic implications of the form $Tgt(Y)$, $Trace(X, Y) \leftarrow Src(X)$ where Src is the source pattern and X the elements matched by it. $Trace$ represents the trace part uniquely mapping source elements X to target elements Y and Tgt the target pattern to be established.

[3] Please note that the object identifiers were chosen to improve readability, but can be completely arbitrary as far as abduction logic programming is concerned.

For instance, the `class2table` transformation rule can be represented as follows:

$$inst(t, \texttt{table}), attr(t, \texttt{name}, n), trace(\texttt{t4c}, t, c) \leftarrow inst(c, \texttt{class}), attr(c, \texttt{name}, n),$$
$$attr(c, \texttt{persistent}, \texttt{true}).$$

Employing ALP requires that there is only one head predicate. Therefore, the target pattern is collapsed into a single predicate and each rule is split in two parts: the source part, which is amenable to abduction, and the target part, which is used to match target patterns. Trace is included in the source part so that changes to it can be abduced. Both parts are discussed in the following sections. For instance, the rule `class2table` can be represented as follows:

Source part: $class2table(t, n) \leftarrow inst(c, \texttt{class}), attr(c, \texttt{persistent}, \texttt{true}),$
$attr(c, \texttt{name}, n), trace(\texttt{t4c}, t, c)$

Target part: $class2table(t, n) \leftarrow inst(t, \texttt{table}), attr(t, \texttt{name}, n).$

4.2 Matching Target Patterns

Transformation rules generally match source patterns and then create the corresponding target pattern. Considering only one application of one rule, say `attr2col`, results in one instance of the target pattern being created. Removing only parts of that pattern–only the column for instance–is not possible. There is no way, the aforementioned rule can produce an isolated column. Only removing (or creating) the whole target pattern constitutes a legitimate target change. Therefore, changes can only be propagated on a target pattern basis. Either the whole pattern is created or deleted.

Usually, target patterns overlap to produce an interconnected network for objects (see Fig. 5), rather than unconnected islands of target pattern instances. For example, consider the transformation in Fig. 3. Assume there is one persistent class with one attribute. Applying the transformation produces one table with the corresponding name through rule `class2table`. Rule `attr2col` (re-)creates the same table and adds a column to it. Through the overlapping of both target patterns, the table is now supported by both rules whereas the column is only supported by one.

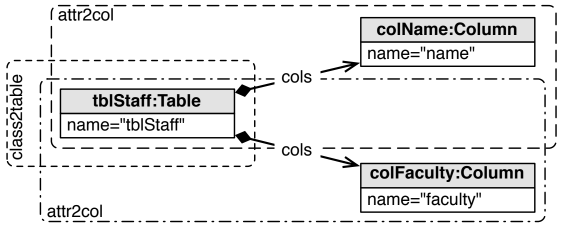


Fig. 5. Object diagram showing overlapping target patterns

This overlapping effectively allows the removal of parts of target patterns, but not arbitrary target elements. For instance, removing the column and retaining the table in the previous example is a legitimate change, even though only a part of the `attr2col` target pattern is removed. The other part, namely the table, is still supported by `class2table`. Changing the source model such that `attr2col` cannot be applied any more results in the deletion of the target instances that were solely supported by this particular rule application. Other target elements, supported by other rules, still remain.

Not only target patterns can be overlapping, but also source patterns. This is also the case in our previous example. The same class is matched by both rules. Therefore, solutions to deleting the column may also accidentally delete the table, which, however, should be retained. Consider a source change where the class is deleted, which results in the deletion of the column *and* the table. To prevent such side effects, overlapping patterns can be actively retained by "observing" their insertion. With respect to the example, deleting the column requires the whole `attr2col` target pattern to be deleted and the table to be retained by (re-)inserting `class2table`.

As changes can only be propagated on a target pattern basis, individual changes need to be coalesced. This is done by matching the target patterns against the old and new target model and requiring that with each pattern at least one change (deletion or insertion) is matched. Based on these matches it can be determined which patterns need to be deleted or inserted. For pattern matches where only some of the elements were subject to deletion, all other elements need to be supported by other rules and actively retained, as discussed before.

4.3 Formulating the Abductive Query

Based on the target patterns that need to be created or removed, the abductive query can be formulated. This step is straightforward for deletions in which case the corresponding source pattern can be looked up in the trace.

For insertions, a more complex process is necessary. Target patterns may be overlapping and even different rules can have the same target pattern. Therefore, it may be possible that not all source patterns can be created to support all target pattern matches. For instance, consider the following transformation:

```
RULE R1                  RULE R2
FORALL X x               FORALL Y y
MAKE Z z FROM f(x)       MAKE Z z FROM g(y)
```

Assume a new instance of Z was created in the otherwise empty target model and this change is now propagated. Both target patterns match this change. However, as they use different uniqueness functions ($f(x)$ in `R1` and $g(y)$ in `R2`) only one of the rules can support the new Z. There is an exclusive choice to be made. Even more complex scenarios are possible where there are two (or more) sets of rules covering the same target patterns. Again, not all of them can and have to support the target patterns.

Essentially, the abductive query is a conjunction of disjunctions of rules, where all variables are bound to target elements. Disjunctions of transformation rules represent the different alternatives. For propagating pattern deletions, rules are negated (e.g. $\neg attr2col(\texttt{tblStudent}, \texttt{colName}, \texttt{`name'})$) to require a set of source changes that explain the absence of this particular pattern match in the new target model. Positively mentioned (not negated) rules lead to explanations in terms of source changes as to the pattern's insertion in the new target model.

Consider our previous example where there was an alternative between rules `R1` and `R2` to support a new instance of `Z`, say `newZ`. The abductive query Q can be formulated as follows:

$$Q_1 = R1(\texttt{newZ}) \vee R2(\texttt{newZ}).$$

In other words, an explanation is sought for the existence of `newZ` either through applying `R1` or `R2`.

Consider the running example (transformation Fig. 3, instances Fig. 4). Assume column `name` in table `tblStudent` was deleted. This results in the deletion of rule $attr2col(\texttt{tblStudent}, \texttt{colName}, \text{`}\texttt{name}\text{'})$. As discussed before, in order to retain table `tblStudent` and column `id` they have to be actively retained by re-inserting their supporting rules into the query:

$$\begin{aligned} Q_2 = & \neg attr2col(\texttt{tblStudent}, \texttt{colName}, \texttt{`name'}) \wedge \\ & attr2col(\texttt{tblStudent}, \texttt{colId}, \texttt{`id'}) \wedge class2table(\texttt{tblStudent}, \texttt{`Student'}) \end{aligned}$$

4.4 Abducing Source Changes

With the abductive query in place, this section now focuses on the computation of the corresponding source changes that provide an explanation to the query. Recall that an abductive framework is a triple (P, A, I), consisting of the program or theory P, the set of abducibles A, and integrity constraints I. Moreover, there is an abductive query Q, which was discussed before and a set of hypotheses H explaining Q.

As part of the program P, a representation of the original (or old) source model is required. As introduced in Sec. 4.1 three predicates are used to encode instances ($inst$), references (ref) and attributes ($attr$) as parts of the source model. A forth predicate ($trace$) represents the trace connecting source elements with their corresponding target elements based on the `FROM`-statements in the transformation. To distinguish between the old and new source model, the aforementioned predicates are prefixed with **old** and **new** respectively.

Since the new source model is not known, it is formulated in terms of changes (insertions **ins** and deletions **del**) to the old source model:

$$\begin{aligned} \textbf{new}\ inst(c,t) \leftarrow & \ \ \textbf{old}\ inst(c,t), \neg\textbf{del}\ inst(c,t) \\ \textbf{new}\ inst(c,t) \leftarrow & \neg\textbf{old}\ inst(c,t), \ \ \textbf{ins}\ inst(c,t) \end{aligned}$$

with similar rules for attributes, references and trace. In words: instances (attributes, references, or trace) are in the new model, if they are in the old and have not been deleted, or if they are not in the old model but have been inserted.

In terms of abductive logic programming, the changes to the source model is the part of the theory or program that is incomplete. It is not known whether a particular instance was inserted or deleted. However, the abductive proof procedure has the freedom to hypothesise about this in order to account for new elements in the target model. The old source model is given and must not be changed. Formally speaking, the predicates prefixed with **ins** or **del** in the above rules are the abducibles in A, which can be assumed as required.

To complete the abductive program P, the transformation has to be included. As introduced in Sec. 4.1 the transformation rules can be written as logic implications, formulated over the predicates that make up the *new* source model. Hence, all predicates referring to source elements have to be prefixed with **new**. As the query Q is formulated over rules rather than individual target elements, only the source side and the trace of the transformation rules are of interest:

$$P = \left\{ \begin{array}{rl} \mathit{class2table}(t,n) \leftarrow & \textbf{new}\ \mathit{inst}(c,\texttt{class}), \textbf{new}\ \mathit{attr}(c,\texttt{persistent},\texttt{true}), \\ & \textbf{new}\ \mathit{attr}(c,\texttt{name},n), \textbf{new}\ \mathit{trace}(\texttt{t4c},t,c). \\ \mathit{attr2col}(t,\mathit{col},n) \leftarrow & \textbf{new}\ \mathit{inst}(c,\texttt{class}), \textbf{new}\ \mathit{inst}(a,\texttt{attribute}), \\ & \textbf{new}\ \mathit{attr}(c,\texttt{persistent},\texttt{true}), \mathit{hasAttribute}(c,a), \\ & \textbf{new}\ \mathit{attr}(a,\texttt{name},n), \textbf{new}\ \mathit{trace}(\texttt{t4c},t,c), \\ & \textbf{new}\ \mathit{trace}(\texttt{col4attr},\mathit{col},[c,a]). \\ \mathit{hasAttribute}(c,a) \leftarrow & \textbf{new}\ \mathit{inst}(c,\texttt{class}), \textbf{new}\ \mathit{ref}(c,\texttt{ownsAttr},a). \\ \mathit{hasAttribute}(c,a) \leftarrow & \textbf{new}\ \mathit{inst}(c,\texttt{class}), \textbf{new}\ \mathit{ref}(c,\texttt{super},\mathit{sc}), \\ & \textbf{new}\ \mathit{ref}(\mathit{sc},\texttt{ownsAttr},a). \end{array} \right.$$

To complete the abductive framework, a set of integrity constraints has to be provided, which is necessary to get sensible answers from the system. These integrity constraints are made up of three parts.

– Constraints concerning the uniqueness of the trace, i.e., there must be exactly one set of source elements giving rise to one target element, regardless of the function used.
– Constraints on the usage of **ins** and **del**, i.e. elements cannot be inserted and deleted at the same time. Moreover, only existing elements can be deleted and only non-existing elements can be inserted.
– Constraints concerning the types of elements, cardinality of references, containment, as derived from the meta-model.

With this set of rules given, the abductive query can be evaluated. This happens in a fashion similar to backward chaining. However, abduction has the freedom to assume the abducibles (the source changes) as required to succeed the query. If the query cannot be succeeded, changes made to the target are not valid and no source changes exist such that the desired target model can be produced by applying the transformation. To avoid littering the source model with excess elements, only minimal source changes are sought.

To illustrate the abduction process, consider the deletion of `name` column in `tblStudent`. As per previous discussions, the corresponding query equates to

$$Q = \neg \mathit{attr2col}(\texttt{tblStudent},\texttt{colName},\texttt{'name'}) \wedge \\ \mathit{attr2col}(\texttt{tblStudent},\texttt{colId},\texttt{'id'}) \wedge \mathit{class2table}(\texttt{tblStudent},\texttt{'Student'})$$

The proof procedure now unfolds the query against P (see previous page) and first tries to fail *attr2col*(`tblStudent`, `colName`, `‘name’`). Therefore, it tries to fail **new** *attr*(`student`, `persistent`, `true`) by assuming the class was non-persistent (**del** *attr*(`student`, `persistent`, `true`)), which is an abducible. Now the integrity checking phase is entered, which declares this solution to be legitimate. However, advancing other parts of the query will rule out this solution as it cannot explain *class2table*(`tblStudent`, `‘Student’`). There are more potential solutions that need to be explored. Another alternative is to fail *hasAttribute*(`student`, `name`) by assuming **del** *ref*(`student`, `super`, `person`). Again integrity check declares this assumption to be viable and indeed it sustains advancing all other parts of the query. The proof procedure will continue unfolding all parts of the query, explore all alternatives and arrive at a set of alternative source changes.

4.5 Compensating Side Effects

Abductive explanations (source changes) may have different qualities when transformed back to the target side. All will explain the observation, i.e., perform the desired target change. However, some will do more and inflict further changes on the target model. Assume that for some reason the `name` column in `tblStudent` is to be removed. The corresponding alternative source changes proposed by the abduction process are:

- $H_1 = \{$delete attribute `name` in `Person`$\}$,
- $H_2 = \{$delete `Person`$\}$,
- $H_3 = \{$delete inheritance relationship between `Person` and `Student`$\}$.

The first two changes have side effects and cause `tblStaff` to lose its `name` column. One solution seems to be to add additional constraints to I that reject these solutions. However, this proves to be too short-sighted as possible solutions may be overlooked. By temporarily accepting these side effects, but requiring their compensation by insisting on reinserting the deleted elements, new solutions can be generated. These new solutions are super-sets of the previous solution. In a new abductive query an explanation is then sought for the observation that `tblStudent` does not have a `name` column any more but `tblStaff` still has.

With this extended query, the system comes up with the following suggestions:

- $H'_1 = \{$delete inheritance relationship between `Person` and `Student`$\}$,
- $H'_2 = \{$move attribute `name` from `Person` to `Staff`$\}$,
- $H'_3 = \{$introduce a new, non-persistent class, move `name` there and make `Staff` a subclass of the new class$\}$

All of these changes exactly perform the requested target change and therefore do not exhibit any side effects when transformed to the target model. Note that we have no basis for choosing any one of these solutions as “superior” but the user might have some criteria unknown to the tool.

4.6 Implementation

In our first attempt to implement the outlined procedure for synchronising models we experimented with ProLogICA [8], which is a direct implementation of the proof procedure in Prolog. It can essentially be applied to any Prolog program. While it was easy to use and the rules needed were essentially the rules outlined above, we quickly run into performance issues and answers took too long to compute, making it infeasible for even quite small transformations and models. Moreover, it required a set of “blank” object identifiers for use in creating new instances of types. This resulted in an combinatoric explosion of isomorphic solutions, where the only difference was the object identifier used to create a new instance of a type.

Our second and current attempt is based on constraint handling rules (CHR) as suggested by Abdennadher and Christiansen [9]. CHR implementations are readily available in most Prolog environments. Given a set of re-writing rules, constraints are rewritten until *false* was produced or no rules are applicable any more. In this case the set of remaining constraints constitute the answer. Models are encoded as aforementioned but in addition are “closed” by a constraint prohibiting the creation of new facts through rewriting. Moreover, modifications to the transformation rules were necessary as CHR does not support negation as failure. Instead explicit negation had to be used, which required splitting the transformation rules in two parts. One for inserting and one for deleting. Even though there was a larger number of rules involved to encode the RTE problem, solutions were produced much more quickly. Moreover, using CHR also allows us deal with attribute value manipulation, such as adding numbers, concatenating strings, etc. and comparisons of attribute values in source patterns. When reversing such rules, a number of constraints can be provided restricting possible values.

Even though the CHR-based approach seems to be far away from how the abductive proof procedure works, it is in fact not that different. Essentially, the same steps are executed but not necessarily in the same order. Queries are still unfolded against the program. This unfolding, however, can be advanced in a more breadth-first manner, building up constraints for explanations quickly, rather than traversing the search space in a strictly depth-first fashion. We believe that this together with the fact that CHR implementations are mature and directly translated, optimised and executed in Prolog, rather than executing an ALP-interpreter, account for the big difference in performance.

5 Related Work

There are a number of existing approaches to model synchronisation and round-trip engineering, imposing different restrictions on the underlying transformations. In general it is required that there is a one-to-one relationship between source and target changes. How this is achieved depends on the concrete approach.

There some approaches centred around a set primitive of injective functions that can be combined to produce more complex transformations. These are guaranteed to be injective and can be easily reversed [10,11]. Injective functions, however, are quite limited and not even arithmetic operations can be used. To lift restrictions on the transformation, Foster et al [12] present an approach based on so-called lenses; pairs of functions defining the forward and the reverse transformation. The forward function solely works on the source model and produces the target model. Conversely, the reverse uses the old source model and the new target to produce the new source model. Still, target changes together with the old source model have to uniquely identify the new target model.

Yet another approach [13] that considers three models for synchronisation is based on triple graph grammars [14], which govern the co-evolution of source and target models. Any relationship—even non-functional relations—can be specified, but not necessarily executed. Changes are propagated by identifying the matching pattern and then establishing or invalidating the corresponding pattern in the other model. When invalidating patterns, all elements are deleted that do not partake in another pattern match. Transformations are not required to be bijective on an element level in order to be usable for this synchronisation approach, as shown by Ehrig et al [15]. However, there must be a one-to-one relationship between source and target patterns.

Fewer restrictions on the nature of the transformation are imposed by the approach presented by Cicchetti et al [16]. It allows for non-injective partial transformations. The reverse transformation, specified by the user, may not be a function and hence there may be several source models for a given target model. However, there is no way to ensure that the provided reverse is reasonable in the sense that when transformed forward again all sources result in the changed target model. Moreover, round-trips without any changes produce all possible source models rather than just the original one.

Query/View/Transformation (QVT) [17] is a recent standard for model transformation, which allows the declarative definition of relationships between source and target models. Relations between models can be checked or enforced in both directions. There is no restriction on the nature of these relationships. They do not have to be one-to-one but can also be many-to-one or even many-to-many in one or the other direction. In other words, there may be more than one source model that corresponds to a given target model and vice versa. This is very similar to the problems discussed in this paper. In fact, QVT model transformation and RTE based on QVT can also be understood as an abductive problem, analogous to Sec. 2.2. We are certain that our technique can also be applied in the context of QVT to support more than just the one-to-one relationships between models.

Not directly related to model synchronisation, is an approach by Varró and Balogh [18] where they propose a model transformation approach based on an inductive learning system. By providing pairs of corresponding models, the inductive logic programming systems derives a set of rules that transform one model into another. As pointed out earlier, there is a close relationship between abduction and induction and therefore, there are also parallels between Varro and Balogh's approach on our approach. However, the premises are different. We consider the theory (transformation) as give and immutable, and derive new source models, as opposed to considering the models as immutable and trying to derive a theory. Self-evidently, both approaches could be combined to result in "synchronisation by example", where the system tries to derive rules based on pairs of source and target changes chosen by a user. This, however, presupposes that all information required to make a choice is contained in the models and the transformation. There may be scenarios where this is the case. As far as our running example is concerned, this does not hold. Whether to delete a class or to mark it non-persistent so as to delete a table is nothing that can be derived from any of the models or the transformation. It rather depends on the meaning a user assigns to the class in question and therefore is not amenable to logic programming.

6 Conclusion and Future Work

In this paper we presented a novel approach to model round-trip engineering based on abductive logic programming. Abductive reasoning, the inference to the best explanation, allows us to compute hypotheses that together with a theory explain an observed phenomenon. We showed how RTE can be interpreted as an abductive problem. Changes to a target model represent the phenomena for which a set of source changes is hypothesised that account for the target changes with respect to the transformation. This was operationalised by translating the transformation into first-order logic with rules of the form "*source pattern* implies *target pattern*". Models were represented using predicates for instances, attributes and references. Abductive reasoning can then be applied to this first-order-logic program to result in a set of source changes performing the desired target change. While performing the target change, the proposed source changes may inflict further changes, side effects, on the target model. In this case compensation can be applied to avoid side effects and arrive at more solutions. The presented techniques are implemented in Prolog using constraint handling rules (CHR), which also allow for dealing with attribute value comparisons in the sense, that solutions may contain a number of constraints restricting attribute values.

With the proposed techniques most of Tefkat's features can be reversed. This includes negation, `LINKS`/`LINKING`s and `PATTEN`s. Features that are not yet supported are recursive `PATTEN`s or reflection. The presented ideas, however are not limited to Tefkat. Also model synchronisation based on QVT or triple graph grammars, which both allow the specification of non-functional relations, can be interpreted as abductive problems.

Abduction was paraphrased as "*inference to the* ***best*** *explanation*" and therefore needs a way to assess the quality of the produced solution. Based on this a list of likely or recommended solutions could then be presented to user or picked automatically. We have investigated very simple heuristics based on the size of the proposed changes, which worked surprisingly well. However, further investigations are required. This heuristic could be directly integrated into the abduction mechanism such that each choice-point creates a backlog of solutions, while only the "best" solution is explored further. Such a merge of abduction and A* search is subject to future work to improve performance and find "good" solutions quickly. Furthermore, Hearnden et al's [19] approach to incrementally propagating source changes to the target model of the transformation, could prove beneficial for incrementally checking for side effects of proposed source changes.

References

1. Hettel, T., Lawley, M., Raymond, K.: Model synchronisation: Definitions for round-trip engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)
2. Pierce, C.S.: Collected Papers of Charles Sanders Peirce, vol. 2. Harvard University Press, Cambridge (1931-1958)
3. Aliseda, A.: Abductive Reasoning: Logical Investigations Into Discovery and Explanation. Springer, Heidelberg (2005)
4. Kakas, A., Denecker, M.: Abduction in Logic Programming. Computational Logic: Logic Programming and Beyond, 402–436 (2002)
5. Kakas, A., Kowalski, R., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation 2(6), 719–770 (1993)
6. Kakas, A., Mancarella, P.: Generalized Stable Models: A Semantics for Abduction. In: Proceedings of the 9th European Conference on Artificial Intelligence, ECAI 1990, Stockholm, Sweden, pp. 385–391 (1990)
7. Lawley, M., Steel, J.: Practical Declarative Model Transformation with Tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
8. Ray, O., Kakas, A.: ProLogICA: a practical system for Abductive Logic Programming. In: Proceedings of the 11th International Workshop on Non-monotonic Reasoning (2006)
9. Abdennadher, S., Christiansen, H.: An Experimental CLP Platform for Integrity Constraints and Abduction. In: Proceedings of FQAS 2000, Flexible Query Answering Systems: Advances in Soft Computing series, pp. 141–152 (2000)
10. Mu, S.C., Hu, Z., Takeichi, M.: An Injective Language for Reversible Computation. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 289–313. Springer, Heidelberg (2004)
11. Mu, S., Hu, Z., Takeichi, M.: An Algebraic Approach to Bi-directional Updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. ACM Transactions on Programming Languages and Systems (2007)
13. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)

14. Königs, A.: Model transformation with triple graph grammars. In: Proceedings of the Model Transformations in Practice Satellite Workshop of MODELS 2005 (2005)
15. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
16. Cicchetti, A., Ruscio, D.D., Eramo, R.: Towards Propagation of Changes by Model Approximations. In: Proceedings of the 10th International Enterprise Distributed Object Computing Conference Workshops, p. 24. IEEE Computer Society, Los Alamitos (2006)
17. Object Management Group (OMG) formal/08-04-03: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification Version 1.0 (November 2005)
18. Varró, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: SAC 2007: Proceedings of the, ACM symposium on Applied computing, pp. 978–984. ACM, New York (2007)
19. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)

Rewriting Queries by Means of Model Transformations from SPARQL to OQL and Vice-Versa

Guillaume Hillairet, Frédéric Bertrand, and Jean Yves Lafaye

Laboratoire Informatique Image Interaction,
University of La Rochelle, France
{guillaume.hillairet01,fbertran,jylafaye}@univ-lr.fr

Abstract. Implementing language translation is one of the main topics within the model to model transformation area. Nevertheless, a majority of solutions promoted by model driven engineering tools focus on transformations related to modeling languages. In this paper, we address query rewriting by means of model transformations. This study has been carried out within the context of implementing an object ontology mapping tool, which could enable bridging object oriented applications and RDF data sources. This approach allows querying RDF data sources *via* an object oriented query which is automatically rewritten in SPARQL (RDF query language) in order to access RDF data. Hence, the developer can freely focus upon the sole application object model. In this paper, we also present with solutions for translating SPARQL queries into object oriented queries, thus allowing the implementation of SPARQL endpoints for object oriented applications.

Keywords: Model Transformations, Query languages, SPARQL, OQL, ATL.

1 Introduction

The Semantic Web [4] envisions the promotion of the Web of Data as a giant interlinked information database [6]; data is expressed and published in a machine-readable format, which enables automatic processing and inference. Semantic Web technologies provide three facilities to manage data, namely: a directed labeled graph model for data representation: RDF [12] (Resource Description Framework); a Web ontology language (OWL [3]) to express data semantics and a query language for RDF (SPARQL [16]) which allows to distribute queries across RDF graphs. These technologies provide a pragmatic way to make the Semantic Web vision operational. Integrating these technologies within an application development process may be valuable in two ways: firstly, Web applications may provide their data in RDF and thus contribute to Web of Data enrichment; secondly, applications could use any available RDF data on the Web and thus enrich their own content (ex: automated data mashup [1]).

One way to achieve this is to allow the manipulation of RDF data as plain objects, thanks to an object/ontology mapping solution, similar to what exists for bridging relational data and object oriented applications. One main common feature of such tools is their dealing with query rewriting. During the implementation of an object ontology mapping, we actually faced the problem of designing a query rewriting tool. This tool should allow the developer to successively write a query according to the application object model, rewrite this object oriented query as a SPARQL query thanks to the previously defined mapping between the object model and an ontology, execute the query on a RDF data source, and translate back the results into objects.

In this paper we study the transformation rules between an object oriented query language and the RDF query language SPARQL. We develop a query rewriting engine on top of an object ontology mapping solution. The query rewriting engine uses the ATL model transformation language [14]. This study demonstrates a novel application area for model transformations.

The remainder of this paper is structured as follows: Section 2 presents related work. Section 3 introduces the context of this work, which is based on the implementation of an object ontology mapping solution performing RDF data access through an object abstraction. Section 4 presents the query languages we chose for this study. Section 5 explains the query rewriting process implementation *via* model transformations. Finally, Section 6 concludes on remarks and future works.

2 Related Work

Providing solutions to expose existing data source content in RDF is of high interest for insuring a real adoption of the Semantic Web. Many approaches have been proposed, mainly about relational to RDF mapping. The W3C RDB2RDF Incubator Group produced a corresponding typology. Other states of the art for database to RDF efforts can be found in [9] and [19]. Some of these approaches provide mechanisms to query relational data with SPARQL.

R2O [18] and D2RQ [5] are based on declarative mapping languages, and can be used to build SPARQL endpoints over RDBMs. R2O provides more flexibility and expressiveness, it needs a referenced ontology. D2RQ directly exposes the database as Linked Data and SPARQL with D2R server. Using either R2O or D2RQ requires an initial learning of the language as well as a good knowledge about modeling. Virtuoso RDF Views [8] stores mappings in a quad storage. While D2RQ and RDF Views follow the table-to-class, column-to-predicate approach, RDF Views has some additional methods to incorporate the DB semantics. All mapping languages have more or less the same complexity and flexibility as SQL. Relational.OWL [11], SPASQL [17] and DB2OWL [9] are other projects that aim to expose relational data on the Web.

SPOON (Sparql to Object Oriented eNgine) [7] addresses the wrapping of heterogeneous data sources in order to build a SPARQL endpoint. The SPOON approach grounds on an object oriented abstract view of the specific source format (as in ORM solutions). SPOON provides a run time translation of SPARQL queries into an OO query language based on the correspondence between the SPARQL algebra and the monoïd comprehension calculus. Our approach shares many conceptual ideas with the SPOON project. Nevertheless, the SPOON approach seems to only operate in case of an ontology that is generated from an object model, and not to offer solutions for complex mappings. Furthermore, SPOON does not propose any solution for rewriting OO queries into SPARQL queries.

3 Context of This Work

In this section, we present the context in which our proposal has been developed. The query rewriting process uses an object ontology mapping solution we are currently developing. This solution allows both RDF data access through an objet abstraction, and conversely ensures an external access to objects *via* SPARQL queries on a corresponding RDF representation. It does alleviate the implementation of SPARQL endpoints.

3.1 Bridging Object Oriented Applications and Semantic Web

The pieces of work presented in this article take place within the context of the development of an object ontology mapping solution. We already presented our main ideas in [10]. We have specified a declarative mapping language and developed a framework that is inspired from object relational mapping framework, but actually deals with bridging applications and Semantic Web sources.

Such a solution allows the publication of data being generated by an object oriented application (here, by a Java application) as RDF data. RDF is dynamically generated and can be used by external applications, taking advantage of the data integration capabilities offered by the Semantic Web technologies. This allows an easier data sharing between applications, for example automated data mashups. The second benefit of such a solution is to allow the developer focusing on the application object model, without having to care about how objects are represented in RDF, and thus not having to define SPARQL queries, but rely upon an object query language instead. One advantage is that the application is no longer bound to a single data source, as usual in object relational solutions.

The framework we developed considers the set of POJO (Plain Old Java Object) classes as the domain object model. POJO classes are represented at runtime as an Ecore model (i.e. a metamodel), thus enabling the representation of object graphs as models conforming to an Ecore model. Considering the application domain model as a metamodel is perhaps unusual, but not penalizing, since classes of the application domain model are supposed to be POJO classes. Such classes generally own properties that can be accessed by getters and setters; Class operations can specify the class behavior. This kind of classes can easily be represented in the form of Ecore models. The use of model enables the execution of model transformations at runtime for both data transformation and query rewriting. Our framework uses the ATL model transformation language.

The object-oriented model proposed by Ecore [20] and the ontology model proposed by OWL [3] share many features. Both include the concepts of classes, properties and multiple inheritances. However, the object model and the ontological model show major differences as reported in [13]. Object model instances are instance of only one class, and must conform exactly to the structure (properties and methods) of their class definition, whereas in RDF/OWL instances may easily deviate from their class definitions or have no such definitions at all. The purpose of our object ontology mapping approach is not to provide a solution to the impedance mismatch between object and RDF. We rather consider ontologies as schema for online RDF data sources that help us to identify required data to be used by an application. The mapping language we have developed helps defining rules for instantiating objects from RDF data, and rules for publishing objects as RDF data. Figure 1 presents the domain object model that will be used as an example throughout this article, as well as the ontology associated with it.

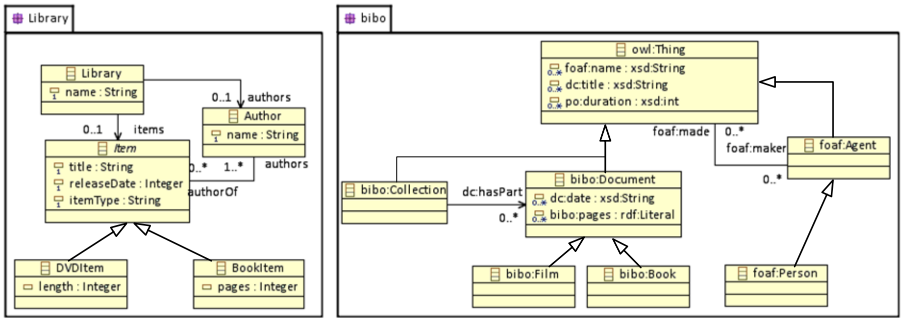


Fig. 1. Domain object model (left) and an excerpt of the bibo1 ontology (right)

3.2 Object Ontology Mapping Language

We propose MEO (Mapping Ecore to OWL), a declarative language for the definition of binary mappings between domain object models and ontologies. This mapping is based on the well-defined semantics of the ATL language. The mapping language is compiled into two ATL transformations, *via* a couple of higher order transformations (refer to [10] for more details). The first generated transformation takes the object model as input, and produces an RDF model (according to the ontology vocabulary). The second generated transformation deals with the opposite way. It takes an RDF model as input and produces a model that conforms to the application domain model, thus finally enabling the retrieval of objects from RDF data sources.

Our mapping language has its own abstract and concrete syntaxes developed by using the TCS toolkit [15]. An excerpt of a mapping is given in Listing 1. The mapping is composed of two mapping rules indicating the correspondences between on the one hand the object model class *Author* and the ontology class *foaf:Person*, and on the other hand, the object model class *DVDItem* with *bibo:Film*. Let's note that two ontologies are used here, since the *bibo* ontology uses terms from the *foaf*[2] ontology.

A mapping rule specifies variables to identify the classes to be mapped. Each variable can be initialized thanks to OCL expressions. The *classMap* operation indicates which classes are mapped, while the *propertyMap* operation maps properties. The *get()* and *put()* operations are others kind of *propertyMap()* operations to be used when complex property mappings are to be defined.

[1] http://bibotools.googlecode.com/svn/bibo-ontology/trunk/doc/index.html
[2] http://xmlns.com/foaf/spec/

```
mapping library
models = {lib : 'http://org.example.library'}
ontology = {
        bibo: 'http://purl.org/ontology/bibo/,
        foaf: 'http://xmlns.org/foaf/0.1/'
}
rule Author2Person {
      def a is lib:Author
      def b is foaf:Person
      def getName is String as b.foaf:name.firstLiteral()
      def putName is String as a.name
      classMap(a, b)
      put(putName, b.foaf:name)
      get(a.name, getName)
      propertyMap(a.authorOf, b.foaf:made)
}
rule DVDItem2Film {
      def a is lib:DVDItem
      def b is bibo:Film
      classMap(a, b)
      propertyMap(a.title, b.dc:title)
      propertyMap(a.length, b.po:duration)
      propertyMap(a.authors, b.foaf:maker)
}
```

Listing 1. Excerpt of a mapping declaration in MEO language

3.3 Specification, Implementation and Execution

The object ontology mapping solution provides a Java library offering common methods for loading/saving objects to/from RDF data sources, as well as a query engine enabling the query of RDF data source through an object oriented query language. This library makes use of models and model transformations at runtime. Using this solution spans three levels:

Specification. The specification step comprises the definition of the object model of the application. This object model is the data model, and is defined thanks to the Eclipse Modeling Framework (EMF) in Ecore. The second specification step is the selection of the ontology that will be mapped to the object model. This ontology can either be a custom one, designed as a simple mirror of the object model, or a common previously existing one, possibly extended by extra concepts. The third specification step is the definition of the mapping between the object model and the ontology thanks to the MEO language. The mapping can be trivial if the object model and the ontology are close to one another, or more complex in case there exist significant differences.

Implementation. The implementation step uses the Java library offered by the object/ontology mapping solution for the Java application which has to access RDF data sources. This library provides methods to save/load objects to/from RDF data sources. In this implementation step, the preferred scenario supposes the generation of POJO (Plain Old Java Object) classes from the Ecore model defined during the specification step.

Execution. The object ontology mapping solution is performed during the execution of the Java application (during loading/saving objects). The execution step uses the ATL engine at runtime. ATL model transformations are generated from our mapping language and executed when translation of objects to or from RDF is required. The ATL engine is also used at runtime for the query rewriting process as presented in this paper.

4 Query Languages

This section presents the query languages used by our approach: an object-oriented language based on OQL and a query language for RDF data (SPARQL). The object-oriented language chosen for this implementation is HQL (Hibernate Query Language) [2].

4.1 Object Query Language

The OQL language is a standard developed by the ODMG (Object Data Management Group) for the expression of queries on object oriented databases. It suffers from too much complexity and has never been fully implemented. Several variants of this standard exist in implementations such as: JDOQL, EJBQL or HQL.

Here, we partially redefine the implementation of the HQL language with the help of MDE toolkits. A metamodel is developed from the grammar of the HQL language. The textual syntax of the language is defined, according to the metamodel, using the TCS toolkit. The execution of a query is made either with the Hibernate framework [2] when needing to retrieve objects form relational data, or by using model transformations joint with our object ontology mapping tool when needing to retrieve objects from RDF data. The latter solution is the one presented in this article.

We limit our presentation to HQL queries of type *SELECT*. The following figure shows the representation of a query of such a type in conformance to the HQL metamodel. A *SELECT* statement is composed of four clauses: *SelectFromClause*, *WhereClause*, *OrderByClause* and *GroupByClause*.

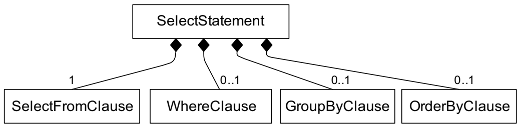


Fig. 2. The Select Statement representation in the HQL metamodel

In a *SELECT* query, only the *SelectFromClause* element is mandatory. Examples of valid HQL queries (according to our HQL metamodel) are given below:

```
(a) from Library
(b) select a.name from Author a, a.authorOf item
      where item.publishDate > '2000'
```

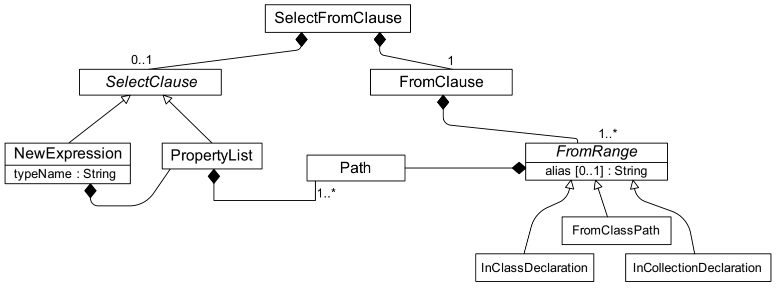


Fig. 3. Representation of the Select From Clause in the HQL metamodel

HQL query results are either lists of objects or lists of values. Query (a) returns the list of objects of type `Library`. Query (b) returns a list of values from the property `name` of all objects of type `Author` having published an item after `2000`.

The *SelectFromClause* representation is given in Figure 3. It is composed of an optional *SelectClause* and a mandatory *FromClause*. A *SelectClause* may be of two types: *NewExpression* is a clause which creates a new object. *PropertyList* contains the list of values or objects resulting from the query. Values are path expressions composed of one or more terms. A term is either an alias for a type of the object model, or a property of the object model. A path permits to browse the object model.

The *FromClause* allows selecting these elements from the object model that will be used in the query. The declaration of these elements is achieved either by indicating the desired class in the object model (represented in the HQL metamodel by *FromClassPath*) or indicating an association. Each element of a *FromClause* is identified by an alias that will be used for specifying paths in the query.

The *WhereClause* element represents the constraint part of the query. A *WhereClause* basically is an expression which can either be a binary expression (and, or), an operator expression (=, <, =<, >, >=), a path expression, or a value (string, integer, boolean).

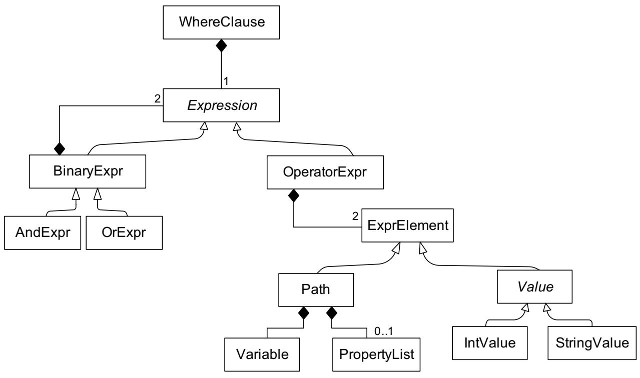


Fig. 4. Representation of the Where Clause in the HQL metamodel

4.2 RDF Query Language (SPARQL)

SPARQL enables the retrieval of data available on the Web in the form of RDF triples. RDF models data as a directed labeled graph that represents the information on the Web. SPARQL can be used to express queries across various RDF data sources, potentially native RDF data, or RDF data generated automatically, e.g. *via* a SPARQL endpoint. A SPARQL endpoint is a server managing the conversion of data (relational or other) into RDF, when answering the receipt of a SPARQL query.

SPARQL is based on the concept of graph patterns for the selection of RDF triples. A pattern is a triple composed of one or more variables. An example of SPARQL query is given below.

```
(c) select ?o where { ?s rdf:type ?o }
(d) construct {
          ?s dc:title ?t ;
             po:duration ?n ;
             bibo:pages ?p .
    }
    where {
          ?s dc:title ?t .
          ?s dc:date ?d .
          filter (?d > '2000') .
          optional {?s po:duration ?n}
          optional {?s bibo:pages ?p}
    }
```

A SPARQL query returns a list of values (*ResultSet*) or an RDF graph. For example, Query (c), of type SELECT, returns a list of values corresponding to the identifier (URI) of the types of all resources in an RDF graph. Query (d) returns an RDF graph constituted by a set of patterns built in accordance to the set of patterns defined in the WHERE clause. Let's note that the WHERE clause in Query (d), includes a test value using a filter, and two optional patterns. An optional pattern is used to select a pattern potentially not present in the graph.

To carry our approach out, we defined the SPARQL metamodel from the SPARQL grammar. We derived the concrete syntax from the metamodel and used the TCS toolkit.

Figure 5 shows the SPARQL metamodel part and addresses the different possible types for a query. In this study, we are only interested in queries of types SELECT and CONSTRUCT.

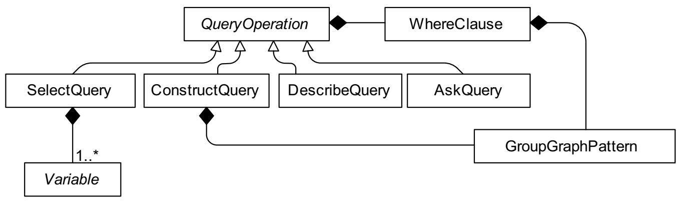


Fig. 5. The different types of Query Operation in the SPARQL metamodel

SPARQL basic concept is the triple. A set of triples forms a graph pattern. The representation of this concept in the SPARQL metamodel is given in Figure 6. The *GroupGraphPattern* consists of a set of graph patterns (*GraphPattern*) representing the various kinds of graph pattern.

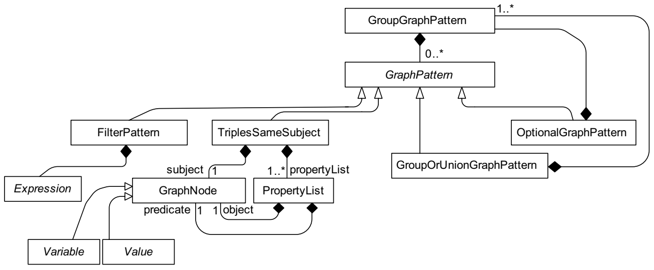


Fig. 6. Representation of the Group Graph Pattern in the SPARQL metamodel

The simpler one is the triple represented by the element *TriplesSameSubject* which includes a subject and one or more associated properties. Other types are *GroupOrUnionGraphPattern* for union of patterns; *OptionalGraphPattern* for optional patterns and *FilterPattern* for specifying filters. Graph patterns are created from nodes. A node is either a variable (named or free) or a primitive type. A named variable is identified by an URI.

5 Model Transformations

This section presents the implementation of our solution for rewriting HQL queries into SPARQL and *vice versa*. We use the model transformation language ATL.

5.1 Rewriting HQL in SPARQL

Rewriting into SPARQL an HQL query expressed in the terms of an object model is carried out by a model transformation that needs two inputs: the mapping defined between the object model and the ontology and the HQL query itself. The following figure depicts the overall rewriting process.

The HQL query is first translated into a model that conforms to the HQL metamodel. This model is used as an input by *HQL2SPARQL.atl*. The output is the corresponding SPARQL model. During the transformation execution, the object ontology mapping is used to identify the correspondences between object model terms and ontology terms. The SPARQL model is sent to a relevant RDF data source. The execution of the query returns an RDF graph. This graph is then transformed into a model conforming to the RDF metamodel. The RDF model is used as input by rdf2model.atl. This transformation is generated by a high order transformation from the object ontology mapping. The execution of *rdf2model.atl* supplies the resulting object graph.

The HQL query returns a list of items or a list of values or a list of values and objects. Hereafter, we detail the first case, and only outline the second. The last case is not yet supported by our implementation. Figure 7 depicts the case where the SPARQL query returns an RDF graph.

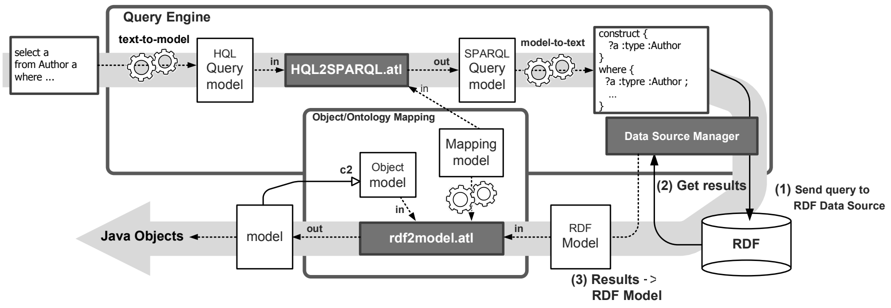


Fig. 7. Query execution process : From HQL to SPARQL

Case 1. The query returns a list of values. In this case the HQL query must be translated into a SPARQL query of type SELECT. This type of query returns a list of values and not an RDF graph. The result of the HQL query is the result of the SPARQL query having been generated. No transformation is needed since the data retrieved actually are plain values.

Case 2. The query returns a list of objects. In this case the HQL query must be translated into a SPARQL query of type CONSTRUCT. This type of SPARQL query returns an RDF graph. The latter will be processed by our object ontology mapping engine, so as to transform the RDF graph into an object model.

The implementation of all cases above is driven by two distinct model transformations. The transformation rules presented below refer to Case 2. However most of the rules are common to both cases, particularly for what concerns the generation of graph patterns from HQL path expressions. Let's explain more about the transformation rules, and let's consider the following query which is defined on the object model presented in Fig. 1 at Section 3.1.

```
(e) select item, author from Item item, item.authors author
     where item.releaseDate > '2000'
```

Let Q be the HQL query, Q' the resulting SPARQL query and M the object ontology mapping. The transformation HQL2SPARQL is hence defined by:

SPARQL2HQL(Q : HQL, M : MEO) → Q' : SPARQL

Rule A: For each *Path* identifying an object belonging to a *PropertyList* element in the *SelectClause* of query *Q (if any)*, a set of triple patterns (*TriplesSameSubject*) is generated and added to a *ConstructQuery* element in query *Q'*.

```
                              construct {
                                 ?item rdf:type bibo:Book ;
select item, author from                dc:title ?itTitle ;
   BookItem item ...      =>            dc:date ?itDate .
                                 ?item foaf:maker ?author .
                                 ?author rdf:type :Author ;
                                   ...
                              }
```

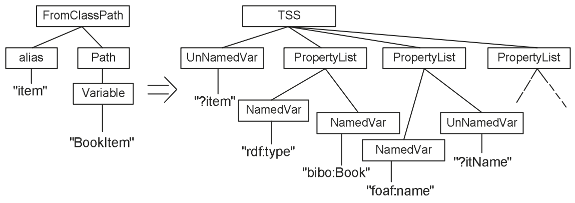
The set of triples enables to retrieve each property belonging to a given RDF property. The set comprises a triple identifying the type of the resource. This type is retrieved according to the mapping. Other triple patterns correspond to the properties required by the mapping.

Rule B: For each *FromClassPath* element from Query *Q*, having a *Path* size equal to 1 and identifying an object, a set of triple patterns (*TriplesSameSubject*) is generated and added to the *WhereClause* in Query *Q'* (cf. figure below).

Rule C: For each *FromClassPath* from Query *Q* having a *Path* element of size *N (N>1)* and being a valid expression path according to the object model, a set of *N-1* triple patterns is created in Query *Q'* by applying recursively the previous rule. (example: `a.b.c => ?a :b ?ab . ?ab :c ?bc`).

Rule D: For each expression in the *WhereClause* of Query *Q* being of type *OperatorExpr and* having the symbol '=' as operator, a corresponding triple pattern is created in the *WhereClause* of Query *Q'*. (example: `s.p = o => ?s :p ?o`)

Rule E: For each expression of type *OperatorExpr* in the *WhereClause* of Query *Q and* having an operator that belongs to the following list: (> | >= | < | =<), a triple pattern and a filter pattern are created in the *WhereClause* of Query *Q'*. The filter pattern contains the expression occurring in the *OperatorExpr*.
(example: `s.p > val => ?s :p ?sp . filter(?sp > val)`)



```
                              where {
from BookItem item ...           ?item rdf:type bibo:Book ;
                                        foaf:name ?itName ;
                                        dc:date ?itDate ;
                                        bibo:pages ?itPages
                              }
```

5.2 Rewriting SPARQL in HQL

Information systems persistency is usually achieved through relational database, XML files, etc… However, using an RDF representation is a way to facilitate data integration and matching, as promoted by the Semantic Web architecture. Semantic Web technologies provide and enable representation of existing data via a common vocabulary (ontologies) that can be extended so that an additional vocabulary could be taken into. Data represented via RDF graphs can be interlinked with each other, allowing an easy navigation within a graph representing the global data on the Web.

Providing tools allowing an efficient and transparent transformation of existing data into RDF is a most important issue Through the abstraction offered by the object model, we can take advantage of mappings, transformations, converters, etc., that already exists between the object abstraction and data sources such as relational, XML, etc. By adding an object ontology mapping tool to these solutions, we can get a complete conversion chain between heterogeneous data sources and Semantic Web data. Query rewriting between SPARQL and an object query language (such as HQL) allows an on the fly generation of RDF data and thus makes it possible to keep data in existing databases and avoid data replication with its entailed lack of synchronization and integrity.

The rewriting process from SPARQL to HQL supposes two prerequisites. First an object model must have been explicitly or implicitly identified (e.g. application classes in the latter case). Second, an ontology is associated to this object model (see example Figure 1). Finally, a SPARQL endpoint has been implemented in order to connect our application to the Web, and thus enable receipt and processing of SPARQL queries.

The SPARQL endpoint is a Web server receiving queries from others Web applications. SPARQL queries received are injected in the form of SPARQL model thanks to a text to model transformation, as depict in figure 8. This SPARQL model is processed by a set of model transformations and result in a HQL model. During those transformations, the object ontology mapping helps to determine how terms used in the SPARQL queries (ontology terms) are converted in terms for the HQL queries (object mode terms). The resulting HQL query is used by a Java application, and may be executed thanks to the Hibernate framework on a relational database. Resulting objects are translated in RDF thanks to our object ontology mapping solution, as depicted by figure 8.

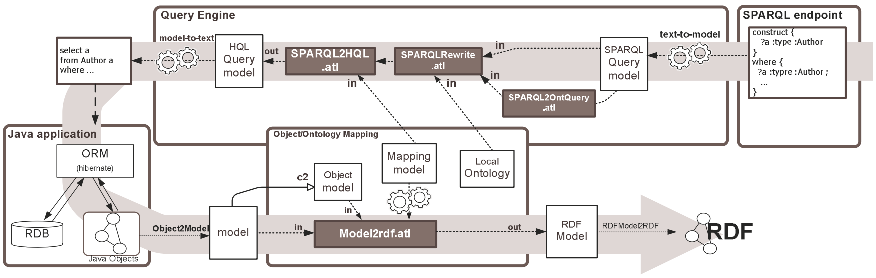


Fig. 8. Query execution process: From SPARQL to HQL

The rewriting process of a SPARQL query into the object formalism chains three steps, corresponding to three model transformations. Let's illustrate our transformation rules, by treating the following SPARQL query as an example:

```
(f)  construct {?a ?p1 ?o1 .?c ?p2 ?o2 }
     where {
       ?a dc:date ?c . filter(?c > '2000') .
        ?a foaf:maker ?b
     }
```

5.2.1 Step 1: Identifying Types in Initial Query

The first step is to identify the types of each variable in the query. All potential types are determined by querying the ontology associated with the object model. For example, Query (f) uses three unbounded variables: ?a, ?b and ?c. By parsing the *Where-Clause* of Query (f), we notice that there is no triple pattern using the RDF property *rdf:type* which would explicitly indicate the type of variables. However, the triple patterns use named variables as predicates (named variables are URIs). So, we can infer the types by identifying the domain and range of the triple pattern predicates. This can be done by simply querying the ontology. To do so, we rewrite the query from the initial one, by adding the unknown types in the select clause. Performing the new query will retrieve the missing information about types. More precisely, the rewriting process is handled by the model transformation *SPARQL2OntQuery.atl* (see Figure 8). For example query (f) is rewritten as follows:

```
(g) select
    ?dateDom, ?dateRang, ?makerDom, ?makerRang
    where {
      dc:date rdfs:domain ?dateDom ;
             rdfs:range ?dateRang .
       foaf:maker rdfs:domain ?makerDom ;
                rdfs:range ?makerRang .
   }
```

The *SPARQL2OntQuery* transformation takes the initial query *Q* as input and is defined as follows: *SPARQL2OntQuery(Q : SPARQL) → Qt : SPARQL*

Running Query (g) over the ontology provides all valid types for the free variables. The result is serialized by a SPARQL Engine implementation (Jena ARQ[3]) in an XML format (SPARQL Results XML Format[4]). A metamodel of this format, marked as SRF, has been defined in order to allow the next step transformation to reuse the former results.

5.2.2 Step 2: Refining the Initial SPARQL Query by Adding Types

The second step takes the previous results into account. Types that have been inferred by performing *SPARQL2OntQuery* are inserted into the initial SPARQL query so as to bind the free variables to their valid types. This is done by the model transformation *SPARQLRewrite.atl* (see figure 8). This transformation takes the initial query *Q*, the ontology *O*, and the result *Rt* produces by query *Qrw* as inputs.

SPARQLRewrite(Q :SPARQL, O :OWL, Rt :SRF) → Qrw : SPARQL

[3] http://jena.sourceforge.net/ARQ/
[4] http://www.w3.org/TR/rdf-sparql-XMLres/

The query *Qrw* is identical to the initial query *Q* except that triple graph patterns identifying variables types are added. This set of additional triples patterns is denoted *Tp*. For each variable v in *Q*, having Type *t* according to *Rt*, there exists a pattern *p* such that p = `{?s rdf:type t}`. Thus the query (f) rewrites as:

```
(h) construct {?a ?p1 ?o1 . ?c ?p2 ?o2 }
    where { ?a dc:date ?c . filter(?c > '2000') .
            ?a foaf:maker ?b ;
            ?a rdf:type bibo:Film .
            ?a rdf:type bibo:Book .
            ?b rdf:type foaf:Person
    }
```

5.2.3 Step 3: Transformation Rules for SPARQL to HQL

The last step encompasses the transformation of SPARQL into HQL. This operation is trivial once the types of the variables are known i.e. when Step 2 is completed. The rewriting process is performed by the model transformation *SPARQL2HQL.atl*. This transformation takes the query *Qrw* and the object ontology mapping *M* as inputs.

SPARQL2HQL(Qrw : SPARQL, M : MEO) → Q' : HQL

The transformation rules address the translation of SPARQL triples graph patterns into HQL path expressions. The main transformation rules are the following:

Rule A: For each triple pattern `{?s` *`rdf:type`* `<URI>}` belonging to the *Where-Clause* in query *Qrw*, a *FromClassPath* (`from` *`ClassName`* `?s`) is created where *`ClassName`* `<- map(<URI>)` and *map* the mapping function from ontology to object model.

Rule B: For each triple pattern `{?s ?p ?o}` belonging to the *WhereClause* in Query *Qrw* and having as predicate an *ObjectProperty* (property with an RDF resource as range), the triple is translated into an *OperatorExp* `(s.p = o)` in query *Q'*. The expression in *OperatorExp* is the *Path* formed by the subject and predicate of the corresponding triple pattern and has for RHS expression its object.

Rule C: For each triple pattern `{?s ?p val}` belonging to the *WhereClause* in Query *Qrw* and *val* an RDF::Literal then the triple is translated into an *OperatorExp* `(s.p = val)` in query *Q'*.

Rule D: For each triple pattern `{?s ?p ?o . filter(?o op val)}` belonging to the *WhereClause* in Query *Qrw* then an *OperatorExp* `(s.p op val)` is created in query *Q'*.

6 Conclusion

In this paper, we presented a query rewriting process implemented by model transformations. These transformations exploit a mapping model that describes the relationship between the elements of a model object with those of an ontology. It was also necessary to define the metamodels of the two query languages handled in these transformations: HQL as an object oriented query language and SPARQL as a graph pattern query language for the RDF data model.

The Model Driven Engineering makes it possible to manage complexity inherent in the translation of requests built on quite different data models. The rules of transformation presented in this paper, implemented using ATL language, include transformations from HQL to SPARQL and opposite directions. The main goal of this work is to facilitate the use and the enrichment of the many collections of RDF data available on the Web without having simultaneously to master the object technologies and Semantic Web technologies.

This work is fully implemented, but it has not been heavily evaluated and so a comparison with others similar approaches, in terms of response time and scalability has not yet been done. Future works include the evaluation of the tool and its extension so as to cope with transformation rules taking more complex query languages features into account, namely *join* for HQL and ‘optional’ and ‘union’ patterns for SPARQL.

References

1. Ankolekar, A., Krötzsch, M., Tran, T., Vrandecic, D.: The two cultures: Mashing up Web 2.0 and the Semantic Web. Web Semantics: Science, Services and Agents on the World Wide Web 6, 70–75 (2008)
2. Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications (2006)
3. Bechhofer, S., Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontology Language Reference. W3C Recommendation 10, 2006–10 (2004)
4. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American 284, 28–37 (2001)
5. Bizer, C., Seaborne, A.: D2RQ: treating non-RDF databases as virtual RDF graphs. In: International Semantic Web Conference ISWC (posters) (2004)
6. Bizer, C., Heath, T., Ayers, D., Raimond, Y.: Interlinking Open Data on the Web Demonstrations Track. In: 4th European Semantic Web Conference, Innsbruck, Austria (2007)
7. Corno, W., Corcoglioniti, F., Celino, I., Della Valle, E.: Exposing Heterogeneous Data Sources as SPARQL Endpoints through an Object-Oriented Abstraction. In: Asian Semantic Web Conference (ASWC 2008), pp. 434–448 (2008)
8. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Proceedings of the 1st Conference on Social Semantic Web. GI-Edition- Lecture Notes in Informatics (LNI), vol. P-113. Bonner Kollen Verlag (2007) ISSN 1617-5468
9. Ghawi, R., Cullot, N.: Database-to-ontology mapping generation for semantic interoperability, 2007. In: Third International Workshop on Database Interoperability, InterDB (2007)
10. Hillairet, G., Bertrand, F., Lafaye, J.Y.: MDE for publishing Data on the Semantic Web, Transform and Weaving Ontologies in MDE (TWOMDE). In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301. Springer, Heidelberg (2008)

11. de Laborda, C.P., Conrad, S.: Bringing Relational Data into the SemanticWeb using SPARQL and Relational. OWL. IEEE Computer Society, Washington (2006)
12. Lassila, O., Swick, R.R.: Resource Description Framework (RDF) Model and Syntax Specification (1999)
13. Oren, E., Heitmann, B., Decker, S.: ActiveRDF: Embedding Semantic Web data into object-oriented languages. In: Web Semantics: Science, Services and Agents on the World Wide Web (2008)
14. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
15. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th international conference on Generative programming and component engineering, pp. 249–254 (2006)
16. Prud'hommeaux, E., Seaborne, A.: others: SPARQL Query Language for RDF. W3C Recommendation (2008)
17. Prud'hommeaux, E.: Adding SPARQL Support to MySQL (2006)
18. Rodriguez, J.B., Corcho, O., Gomez-Perez, A.: R2o: an extensible and semantically based database-to-ontology mapping language. In: SWDB (2004)
19. Rodriguez, J.B., Gomez-Perez, A.: Upgrading relational legacy data to the semantic web. In: Carr, L., Roure, D.D., Iyengar, A., Goble, C.A., Dahlin, M. (eds.) WWW, pp. 1069–1070. ACM, New York (2006)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading (2008)

Solving Constraints in Model Transformations

Andreas Petter, Alexander Behring, and Max Mühlhäuser

Technische Universität Darmstadt, Department of Computer Science,
Telecooperation, Hochschulstr. 10, D-64289 Darmstadt, Germany
{a_petter,behring,max}@tk.informatik.tu-darmstadt.de

Abstract. Constraint programming holds many promises for model driven software development (MDSD). Up to now, constraints have only started to appear in MDSD modeling languages, but have not been properly reflected in model transformation. This paper introduces constraint programming in model transformation, shows how constraint programming integrates with QVT Relations - as a pathway to wide spread use of our approach - and describes the corresponding model transformation engine. In particular, the paper will illustrate the use of constraint programming for the specification of attribute values in target models, and provide a qualitative evaluation of the benefit drawn from constraints integrated with QVT Relations.

Keywords: model driven engineering, constraint solving, constraint programming, model transformation.

1 Introduction

Declarative model transformation languages have gained a lot of attention in the model transformation community, especially by discussing and proposing transformation languages like QVT [1]. They are deemed to have several advantages [2]. Some declarative approaches follow the relational approach, which could be called "logic programming for model transformations". These languages relate model elements from the source model to model elements of the target model.

Some models benefit from the use of constraints to overcome possible underspecification. Therefore, constraints have seen widespread usage in practical modelling [3,4]. Furthermore, constraints have shown their declarative nature for programming tasks. Freuder even states that constraint programming would yet be the closest approach to finding the holy grail of programming [5].

Models with constraints may be transformed using standard model transformation languages. But, as transformation languages are not able to cope with constraints to specify aspects of target models, transforming these models will not take into account the constraints contained in the models (or meta-models) directly. Thus, if developers of transformations use model-to-model transformation languages and want a constraint based problem to be satisfied in target models, they will need to solve the constraints by hand after the transformation has finished.

Our contribution therefore is the *combination of constraint programming and model- to-model transformations*. Hereby we fill the gap how constraints can be specified in model transformations and how they can be satisfied in target models. Due to the generality of the basic approach, called "Constraint Relational Transformations" many model-to-model transformation languages can benefit from our contribution.

The contributions in this paper are:

– a model transformation language to define model transformations with constraint satisfaction problems,
– using constraint programming for the specification of attribute values in target models,
– a qualitative evaluation how well constraints integrate with declarative model transformation languages, investigated on the example of QVT Relations, and
– an implementation of a model transformation engine which is able to solve constraints.

1.1 Overview

The third section defines "Constraint Relational Transformations" which allow for constraint solving to be used in transformations. Section 4 presents how we changed QVT Relations and OCL to support for constraint solving. Section 5 presents our implementation of the transformation engine while section 6 discusses the benefits and 7 the state of the art. Section 8 summarizes the findings and presents opportunities for further research.

2 Constraints and Running Examples

It is important to note that the complexity of the constraint language greatly affects the complexity of the algorithms needed to satisfy the constraints. For example it is comparatively easy to solve continuous linear constraints, but difficult to solve discrete, non-linear constraints. However, in model driven engineering the most frequently used constraint language is the Object Constraint Language (OCL) [4]. It allows for the definition of constraints on almost any facet of common model elements. The language has been found to be non-computable in theory [6]. Therefore, it imposes a problem to do constraint solving over all aspects of the OCL language. Constraint solving over OCL constraints can only be done efficiently on a subset of the language. While it remains an interesting research task to formally identify this subset, we restrict our constraints to the subset of non-linear numeric constraints. This type of constraints is needed in many domains where constraint programming can be applied [7].

Constraints can be divided into *two different types: constraints over attribute values and associations in transformations versus global graph constraints*. The first type of constraints can be attached to model elements and therefore are *"local"* to them. Nevertheless, they can span several model elements through transitive use of associations. Constraints used in common modeling languages are local (e.g. UML and OCL). We assume that most constraints used in modelling fall into this category. The second type is global to a transformation and requires the transformation engine to satisfy them even across all model elements. Scheduling and planning problems fall into this category. We only take into account constraints of the first type. The second type would require extensive research how the concept of graph constraints can be combined with a model transformation language and integrated with constraints over attributes and associations.

This paper focuses on numeric, non-linear, local constraints. Hereby, problems like generating user interface models or problems from operations research can be addressed using a declarative model-to-model transformation language with constraint solving.

Our constraint based approach can also be used to simultaneously span constraints over source and target models at once. This is not our research focus as the main application of such a setup would be model synchronization, which we are not heading for. Therefore, the language presented can only use variables to retrieve information from the source model and not use constraints which simultaneously enforces constraints on source and target model elements.

2.1 Transformation Examples

Even in very simple model transformations numeric constraints can be a requirement. Our illustrative example is borrowed from the user interface community where constraints have been widely used to define user interfaces. The developer wants to develop a model-to-model transformation that is able to account for adaption to the display of the target platform. The target platform will show a window (c.f., figure 1), which contains three standard buttons ("Ok", "Cancel", and "Ignore"). The suitable meta-model of the abstract user interface model is shown in figure 2 while the platform specific one is shown in figure 3. The sizes of the components used when displaying are retrieved from the model (e.g. because they can be altered in a graphical model editor). Therefore, the model-to-model transformation should set them.

Hereby the following constraints must be satisfied: The sum of the width of the buttons should not be larger than the window and the window should fit on the screen (800 pixels). Furthermore the buttons must have minimum and maximum sizes (e.g. $30 \leq \text{width} \leq 60$). The resulting model transformation will be a constraint solving problem that transforms an abstract user interface model into a platform specific user interface model.

This example currently can not be executed using a model-to-model transformation language without investing significant effort.

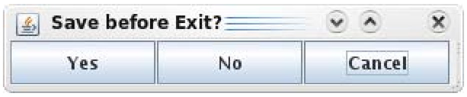


Fig. 1. Outline of the desired target window for the example (JAVA Swing)

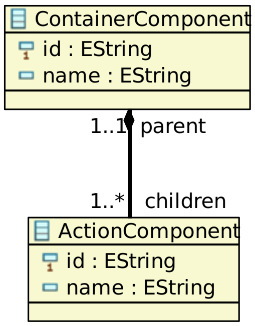


Fig. 2. Meta-model of the abstract user interface model

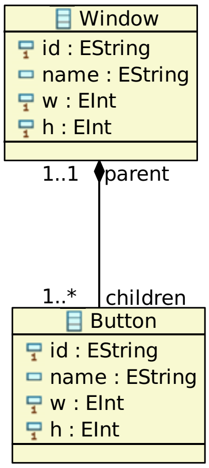


Fig. 3. Meta-model of the platform specific user interface model

To implement the transformation with a common model-to-model transformation language, the developer would need to either write a specialized constraint solver using imperative language constructs, which is cumbersome, or use a constraint solver library, pass the constraints to the solver using a special language and call it from the transformation. Obviously, both approaches require efforts, which are not to be underestimated and may be different for each type of transformation being implemented.

3 Constraint Relational Transformations

Depending on the definition of the term "constraints", different mechanism to reflect and solve the constraints need to be applied in transformation languages.

Especially, it must be made clear what type of constraints must be supported. If constraints are restricted to equations, then QVT Relations will classify as a constraint solving approach [8]. While one can arguably call an equation a constraint (e.g. PropertyTemplates in QVT Relations are equations), in fact no constraint solving is performed. Instead, concepts from logic programming are applied. Czarnecki mentions logic programming [8] as the standard way to implement such approaches. Hereby, the main concept is "unification" which is able to cope with variables used in predicates. However, logic programming has been extended to constraint logic programming to support constraints used by constraint solvers (e.g. numeric constraints). In analogy, we extend relational transformations to "constraint relational transformations".

Constraint relational transformations support local or global constraints, as has been motivated in the last section. Hereby, local constraints can be applied on values of model elements of the target model to enforce relations between them. Additionally, variables may be used to relate source model elements to target model constraints. A set of constraints forms a constraint system, called CSP. This is a constraint program, which needs to be solved by a constraint solver. For example, two model elements representing user interface components are contained in a model element representing a container component. The model elements are related, because they share the same parent space on the screen but may not overlap. Therefore their sizes add to the size of the container component (see section 2.1). This is normally done by using numeric constraints. Satisfaction of these numeric constraints in target models can be established using constraint solving and in the case at hand should be done in the model-to-model transformation engine.

4 Extending the Syntax of QVT and OCL

Defining a completely new transformation language is difficult, especially when it should be declarative. Therefore this section enhances a concrete transformation language to become a constraint relational transformation language. QVT Relations is known to have a very declarative and abstract notation [9]. Despite commonly known problems (e.g. [10]) we chose QVT Relations because of its declarativity, which we believe is a good start to add concepts from constraint programming. However, extending QVT Relations is only an example and the main concepts presented here, can be used to extend other relational model transformation approaches, e.g. Tefkat [2] or MTF [11], as well.

Our QVT Relations dialect is called "Solverational". To implement Solverational we needed to change the syntax of QVT Relations, as well as the semantics of OCL.

4.1 QVT Relations

Transformations written in a QVT language are defined over MOF [12] based meta-models. MOF may be seen as to be a small subset of the well known UML,

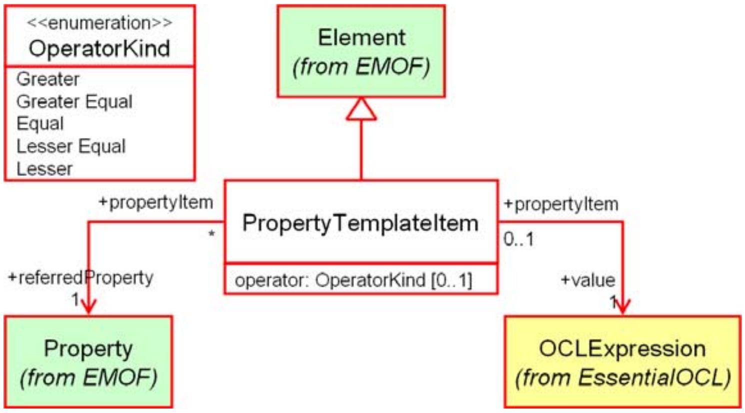


Fig. 4. QVT Relations abstract syntax of PropertyTemplateItem. The PropertyTemplateItem is extended with the OperatorKind attribute to support inequalities.

which is usually used to define meta-models. A transformation rule in QVT Relations is called a "relation". Simplified, a relation is a mapping between several sets of model elements, called "domains". Each domain is based on a meta-model class. The domains of the model elements are defined by types from the source and target meta-models.

Every domain may contain "PropertyTemplateItems". If applied to the source domain, the PropertyTemplateItems will select instances from the model. In the target domain they set the properties or associations of the target instance. These template items will use values from other domains, if their variable names are declared within the relation. *PropertyTemplates in traditional QVT Relations cannot contain inequalities or multiple constraints to assign values*, instead equations are used.

We defined a simple extension to the QVT Relations textual concrete syntax [1]: *The equal sign in "propertyTemplate" may be replaced by smaller, greater, smaller or equal, or greater or equal.* The following grammar rule is a replacement for the one given in the QVT Relations definition. This simple extension already has significant implications on the model transformation process. Current model-to-model transformation engines are not able to execute a transformation based on this grammar.

```
<propertyTemplate> ::=
    <identifier> ('='|'<'|'<='|'>='|'>'|'<>') <OclExpressionCS>
```

The complete QVT Relations abstract syntax meta-model can also be found in [1]. In figure 4 the meta-model element PropertyTemplateItem is depicted, a small part of the QVT Relations meta-model. A PropertyTemplateItem has a property (which is actually the left side of the equation in the traditional concrete syntax) and an OCLExpression. To allow for more than equality in

PropertyTemplateItems we introduce comparator types (the name "operator" is being used in the OCL description, so we used it in the model as well).

Due to missing definitions in the original QVT Relations document, the semantics need to be made more precise, only. The QVT Relations specification states that the "property value must match the ... expression". As the word "match" is not defined we assume that it is understood as "equality in at least one item of a set". We refined the meaning of "match" and suggest that a match also occurs in the case the property value is really unequal ("$<$", "$>$", "$\leq$", "$\geq$", "$\neq$").

4.2 OCL

Our constraints heavily rely on aggregation functions to evaluate constraints over associations. In OCL aggregation functions (e.g. sum, max, min, product) are used on collections. To support most intuitive usage in the transformation language these aggregation functions were derived from the ones used by OCL but may be used directly in OCL expressions. Because they are not used on collections, their signatures have changed. We implemented only a subset of the OCL, that is most interesting for constraint solving (numeric constraints).

The constraints may have arguments, which are evaluated by identifying appropriate model elements and their values in the model. They may refer to associations to access other model elements. Results from the aggregation functions are not directly computed, but used in the constraint solving process to determine the values of all model elements and their attributes involved in the constraints.

This can best be explained by referring to the Solverational description of the example, which is shown in figures 5 and 1. The function "sum" sums up all "x" attributes of the model elements associated by the "childs" association. However, the value of the sum is not pre-determined. In fact, the transformation will calculate appropriate values for the x attribute by relating the values of the children. We call this form of transformation containing smaller equal constraints "smaller-equal-transformation".

We also wrote a different version of this transformation with a pre-determined sum. The width "w" of the model element called "Window" is fixed by setting it to $w = 800$. This forces the constraint solver to determine values for the "w" attributes of the child elements. This form is referred to by "equal-transformation".

4.3 Additional Solverational Features

The most obvious strength of Solverational is its ability to use constraints instead of using attribute assignments. Therefore the transformation developer is freed from the cumbersome task of solving the constraints by hand afterwards. In Solverational this can be noted in an intuitive way. The developer simply uses inequality comperators to get a solved target model. Every attribute can be subject to a multitude of constraints. Therefore - in contrast to QVT Relations, where this does not make sense (ironically it is not forbidden in the specification), the developer can use the same attribute multiple times in the same rule (the same "ObjectTemplateExp"). Each time he may specify a different constraint

```
transformation trafo (a:SemiAbstractUIMetaModel, b:ConcreteUIMetaModel) {

top relation SACopm2CComp  {
   theID : String;
   theName :String;
   theChID : String;
   theChName :String;
   domain a c:SemiAbstractUIMetaModel::ContainerComponent {
      id = theID,
      name = theName,
      children = f:SemiAbstractUIMetaModel::ActionComponent {
         id = theChID,
         name = theChName
      }
   };
   domain b d:ConcreteUIMetaModel::Window {
      id = theID,
      name = theName,
      w = sum(children.w) + 2*(count(children) - 1) + 6,
      w <= 800,
      h = max(children.h) + 13,
      h <= 600,
      children=g:ConcreteUIMetaModel::Button {
         id = theChID,
         name = theChName,
         w <= 60,
         w >= 30,
         h <= 20,
         h >= 10
      }
   };
}
}
```

Fig. 5. Transformation example noted in Solverational

on the attribute. The attribute may also take part in a computation which will implicitly enforce a constraint on the attribute value, if the result of the calculation is set by a different constraint. If the developer over-specifies a property, such that the constraints are not solveable at all - the transformation engine will not transform the transformation at all and report failure to do so.

Solverational allows for usage of aggregation functions to sum, multiply or select elements in a set (OCL terminology: "collection"). As Solverational is able to do work on target models and not just copies values, this adds a new form of relationship to transformation processes: attributes of target model elements can be related in the transformation rules. Even though this is not possible with normal programming languages, it is intuitive as it is a completely declarative way to express this class of complex relationships.

5 Implementation

After introducing the concept of constraint relational transformations in section 2 and an extension to QVT Relations, this section presents a prototype

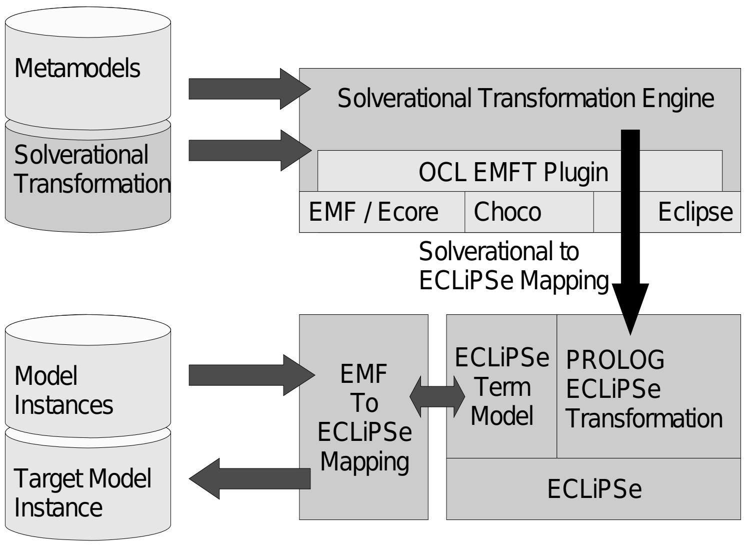


Fig. 6. Architecture of the transformation engine

implementation. The architecture presented in figure 6 can be devided into two parts. *The upper half of the figure shows the compilation step.* From the meta-models of the source and target models together with the Solverational transformation, an ECLiPSe form is produced (transformation compiler). This program can be used for an arbitrary number of transformations of different model instances (transformation compiler). The form depends on the direction of the transformation and must be regenerated in case the direction changes. Then, *the compiled transformation is executed as depicted in the lower half of the figure.* For the execution the models are read-in and transformed by the ECLiPSe form.

Compilation step. The language for the compiled transformation has been chosen to best support the mapping process from Solverational (this was inspired by the verification from Cabot et al. [13]). As a result from the thoughts given in [14] we chose a logic programming language. This language needs to support constraint solving, as is done by the chosen ECLiPSe[1] PROLOG dialect ([15]). ECLiPSe is a constraint logic programming language, which supports common constraint solvers and search algorithms. It allows for the definition of heuristics and the search order of domain values.

The meta-model and the Solverational description is used to produce the ECLiPSe representation of the Solverational transformation definition. Several challenges need to be addressed in this compilation step.

[1] The ECLiPSe PROLOG dialect is used for constraint programming. The Eclipse IDE is used for developing JAVA applications. We used both and created a plugin for Eclipse to generate ECLiPSe model-transformation-programs.

For example, the execution order of the relations can be seen as a scheduling problem (a Solverational transformation can contain multiple relations). This problem can best be addressed with a constraint solver. Using the constraint solver "Choco" ([16]), the Solverational model transformation engine orders the relations according to "when" and "where" clauses. The result is used to write the relations in a correct execution order into the ECLiPSe PROLOG file. Thus, at runtime, relations will automatically be executed in correct order. However, we never tested this approach with a very large number of transformation rules.

Execution step. The EMF to ECLiPSe mapping transforms EMF models into a term based representation in ECLiPSe. Every meta-model element is identified by a term. The terms of the nodes, attributes, and references have special notations which can be distinguished according to their type. When transforming an EMF model, the mapping component creates identifiers and stores them until the transformation process is completed. Afterwards it can map instances of model elements back onto their corresponding EMF representation or create new ones. The transformation of the models to its ECLiPSe form as well as the execution of the transformation in ECLiPSe are performed automatically from JAVA using embedded ECLiPSe programs. During the execution the algorithm given in section 5.1 is executed.

The transformation engine has been realized as an Eclipse plugin, because the Eclipse IDE provides support for many modeling tasks, such as loading and saving models which are used by the engine. The developer can therefore combine the transformation plugin with many of the modeling tools developed by many of the Eclipse projects.

5.1 Algorithm

A generated program that implements a mapping from Solverational to ECLiPSe is a sequence of 6 steps. Their purpose is to collect constraints which need to be satisfied and solve the generated CSP (constraint solving problem). Each CSP variable maps to an attribute in the target model. As soon as the CSP finds a solution, the values of the attributes will be assigned from the CSP variable.

An important aspect is the question at what point a constraint can be inserted in to the CSP. If associations are used in a constraint it can only evaluated after all relevant model elements have been created. Therefore, our generation is composed of 6 steps:

1. First, target model elements are being created or searched for.
2. When a model element is created or found, the algorithm creates a variable for each attribute of the model element on the heap, so they can be resolved during later steps.
3. If there are constraints for the attributes of the model element they are inserted directly into the CSP.
4. After all model elements have been created, constraints for attribute values spanning associations are inserted.

5. Then, all variables of the attributes are retrieved from the heap and inserted into the CSP.
6. Finally, the CSP is going to be solved by executing the constraint solver and results are calculated.

Our algorithm can also cope with the selection of a specific target model out of the set of possible target models in an intuitive way by defining a target function over the set of model elements. However, this is a whole work on its own and due to space constraints we must restrict ourselfs to the contribution of constraint programming in model transformations.

6 Discussion

Our example shows that developers can benefit from the declarative style of constraint programming. We assume that the size of the effect is domain dependent. Nevertheless, as this seems to be the case for many domains [7], we conclude that constraint programming may be a valuable asset to model transformation.

Implementing even the very simple example using the plain QVT Relations language, instead of Solverational, is very difficult. The only option would be to specify values by hand after the transformation has been performed. However, this can only be done for a small set of model elements and it has to be done every time the transformation is executed on a different set of input models. This is cumbersome and a clear indication that this process should be automated. The concepts presented in this paper allow for automating transformations using local constraints.

The implementation shows that it is feasible to do constraint solving on smaller models. To determine the size of models that can be handled efficiently, we evaluated three transformations on models with increasing numbers of model elements. As explained in section 4.2, the three Solverational transformations use different constraints. The "smaller-equal-transformation" constraints the maximum width "w" of "Window" model elements, while the "equal-transformation" sets the size to a specific value and enforces "w" of children to be calculated by the constraint solver. Of course, we had to alter the maximum size constraint for "w" of "Window", such that it still fits the sum of model elements of the "w" of the "children", depending on the number of model elements we wanted to transform.

Additionally, we tried to run the transformation without any constraints (because plain QVT is not able to solve constraints) with an open source QVT Relations transformation engine ("QVT medini" [17]). We expected it to be much faster, as it does not use constraint solving and is an industrial product. It is noteworthy that Solverational is a QVT Relations dialect and therefore is compatible with a large subset of QVT Relations transformations.

We measured the CPU time for the different transformations. We did not add the time needed to save and load the models, but only the time needed

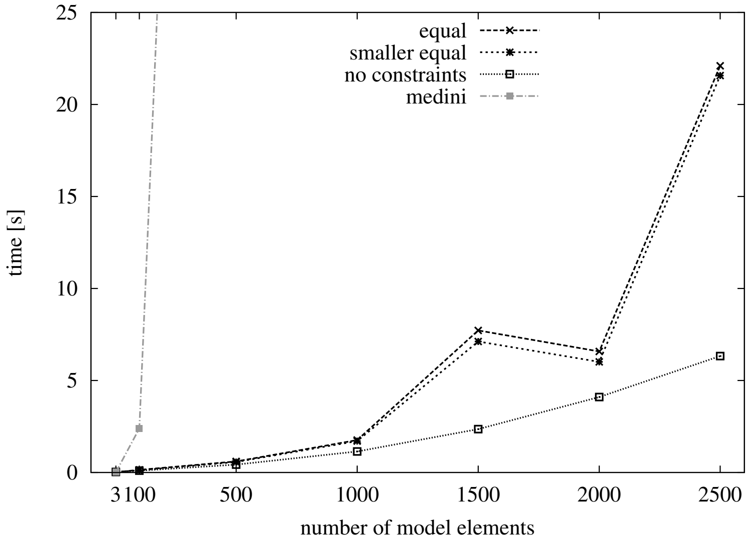


Fig. 7. Number of model elements versus time in seconds for different transformations

to perform the transformation. All results were taken on a computer with a 1.8GHz Intel CPU and 1GB RAM, which was not exceeded during the test runs. Reperforming several transformations on other CPUs indicate that the time needed to perform the transformations is proportional to the speed of the CPU. The results are illustrated in figure 7.

Execution times of the Solverational transformation engine increased as expected, with a slight decrease of calculation time by enforcing constraints for 2000 model elements. The constraints which were more difficult to solve took more CPU time. The "equal-transformation" took longest to compute, because the constraint solver needed to set the values of the "children" model elements. Although, time increased only slightly compared to the one needed for the "smaller-equal-transformation".

Both transformation engines seem to have sub-exponential execution behaviour, but to our great surprise our transformation engine seems to be comparatively fast. Note, that in the case of 2500 model elements, the constraint solver needs to solve more than 10000 constraints with more than 5000 variables. But still the medini QVT engine had considerable larger execution times.

These results indicate that using constraint solving in model transformations is not the time-killing factor. The medini QVT engine implements other features, which seem to be far more time consuming than constraint solving in the model transformation process. However, our examples used very simple constraints and solving time is highly dependent on them. Using e.g. non-linear constraints can arguably produce completely different results. However, we argued in section 1 that many modeling constraints are similar to the ones we were using.

7 Related Work

There are two categories of related work: work on model or graph transformation and work on models and constraint solving.

Ehrig et al. outline that constraint solving is a standard way to optimize pattern matching to search for model elements in model transformations [18]. This has been used in Attributed Graph Grammars (AGG) [19]. However, searching differs from our constraint solving for target models, because our approach enforces constraints in target models.

In different work Ehrig provides a graph grammar based theory which shows how graph constraints can be transformed into application conditions of transformation rules [20]. Then the application conditions support the consistency of graphs involved. The graph transformation rules may not be fired when an application condition (and therefore a graph constraint) is violated. This is not equal to constraint solving, where the solver enforces attribute values.

El-Boussaidi and Mili present an approach that is able to search for model patterns [21]. It is based on the ILOG JSlover constraint solver library and searches for patterns by solving a constraint solving problem over graphs, similar to the approach mentioned above [19].

A proposal called "xMOF" [22] to the OMG RFP for the QVT [23] transformation language has not been accepted as such, but the approach presented uses constraints within the definition of the model transformation. The approach recommends to use a minimalistic extension to OCL [4] to develop model transformations based on meta-models written in MOF [12]. The proposal has not been implemented, so it does not use constraint solving (OptimalJ, for which xMOF had been designed, implements QVT Core instead). From our experiences with our current implementation of the transformation language we suspect that an efficient implementation of the proposal would be rather difficult.

Lengyel et al. allow for the definition of transformation rules and the attachment of constraints [24]. They define "constraint validation", "constraint preservation" and "constraint guarantee" for model transformations. However, the OCL constraints are not enforced by their implementation and therefore does not use constraint solving. As they do not use backtracking the approach cannot produce all possible solutions. Furthermore, it uses XSLT which cannot cope with constraint satisfaction problems.

White et al. focus on using constraint solving to weave models [25]. Model weaving is similar to model transformation in that it is able to weave (or transform) several models into a single model. However, this model must then be transformed by model transformation to be in accordance with the output target platform meta-model, which is a concrete weaving model. However, White et al. do not provide a transformation language that performs constraint solving, but a model weaver which is integrated in the GME environment. Therefore, the constraints of the target platforms would need to be integrated into the constraints of the source model or the model weaver, which would not be very useful, since they should be abstract to produce an abstract weaving solution.

Our model transformation language can be applied to complement the solution in the transformation process.

Constraints have also been used to transform models of user interfaces. SUPPLE [26] uses constraints to define the properties of target platforms. The transformation does not transform the user interface descriptions into models, but the user interface is directly displayed after the constraints are solved. Furthermore SUPPLE does not use a model-to-model transformation language to define the mapping between model elements and the user interface. Therefore, the model can not be modified by the developer, before it is being displayed.

MASTERMIND uses declarative models to construct software based on generative constraints to enforce dependencies between user interface components [27,28]. Starting from the MASTERMIND textual format, model-to-code transformations can be executed, which take into account layout constraints given in a presentation model. MASTERMIND does not use a model-to-model transformation language and therefore is not able to handle constraints in model-to-model transformations.

In a limited scope our approach is also well suited for round-trip engineering as it is able to produce a set of possible source models when it is used as a bidirectional transformation. A definition of round trip transformations and how they relate to the problem of multiple possible source models is given in [29]. In that sense our approach is therefore similar to the one presented in [30], which also uses a logic programming language to implement the generation of multiple source models. However, while Cicchetti's approach is used mainly for round trip transformations and does not use constraint solving but answer set programming to produce possible target models, our approach focuses on the transformation of consraints on attribute values.

8 Conclusions and Future Work

In this paper we presented how constraint solving and model transformations are integrated to constraint relational transformations. Constraint relational transformations allow for the first time to solve local numeric constraints during model-to-model transformations. It was shown that constraints integrate, almost naturally, in QVT Relations transformations. The new dialect is called Solverational.

We presented our implementation of a model transformation engine and showed how a model transformation engine can be extended to support constraint relational transformations. We showed that our concept, when applied, is not the main factor for transformation times.

Our goal is to use the transformation language on user interface models. This will allow for optimization of the user interface models during model-to-model transformation. This work will be applied in the domain of crisis management within the SoKNOS project.

Further, it is planned to work on the integration on graph constraints that can be used for planning and solve problems known from operations research.

This will open up more complex application scenarios to model-to-model transformations suggesting constraint solving, e.g. planning and scheduling.

Acknowledgements. The work presented was partly funded by the SoKNOS project. The funding is provided by the German Ministry of Education and Research (BMBF).

References

1. OMG: Meta object facility (mof) 2.0 query/view/transformation specification. OMG, ptc/07-07-07 (July 2007)
2. Lawley, M., Raymond, K.: Implementing a practical declarative logic-based model transformation engine. In: SAC 2007: Proceedings of the, ACM symposium on Applied computing, pp. 971–977. ACM, New York (2007)
3. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Object Technology Series. Addison-Wesley, Reading (2003)
4. OMG: Object constraint language omg available specification version 2.0. OMG (May 2006)
5. Freuder, E.C.: In pursuit of the holy grail. Constraints 2(1), 57–61 (1997)
6. Brucker, A.D., Doser, J., Wolff, B.: Semantic issues of OCL: Past, present, and future. Electronic Communications of the EASST 5, 213–228 (2006)
7. Ratschan, S.: Applications of quantified constraint solving over the reals. Internet (January 2008), http://www2.cs.cas.cz/~ratschan/appqcs.html visited 01/09
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
9. Jouault, F., Kurtev, I.: On the architectural alignment of atl and qvt. In: SAC 2006: Proceedings of the, ACM symposium on Applied computing, pp. 1188–1195. ACM Press, New York (2006)
10. Stevens, P.: Bidirectional model transformations in qvt: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
11. IBM United Kingdom Laboratories Ltd., I.a.: Model transformation framework (mtf). IBM alphaWorks (2004), http://www.alphaworks.ibm.com/tech/mtf
12. OMG: Meta object facility 2.0 core final adopted specification. OMG (October 2003)
13. Cabot, J., Clariso, R., Riera, D.: Verification of uml/ocl class diagrams using constraint programming. In: Model Driven Engineering, Verification, And Validation: Integrating Verification And Validation in MDE, MoDeVVA 2008 (2008)
14. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of mda. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 90–105. Springer, Heidelberg (2002)
15. Apt, K.R., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press, New York (2007)
16. Jussien, N., Rochart, G., Lorca, X.: The choco constraint programming solver. In: CPAIOR 2008 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP 2008), Paris, France (June 2008)
17. ikv++ technologies AG: Qvt medini. Internet, http://www.ikv.de/ikv_movies/mediniQVT.swf

18. Ehrig, K., Taentzer, G., Varro, D.: Tool integration by model transformations based on the eclipse modeling framework. Technical report, EASST Newsletter (2006)
19. Rudolf, M.: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 381–394. Springer, Heidelberg (2000)
20. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Constraints and application conditions: From graphs to high-level structures. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 287–303. Springer, Heidelberg (2004)
21. El-Boussaidi, G., Mili, H.: Detecting patterns of poor design solutions using constraint propagation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 189–203. Springer, Heidelberg (2008)
22. Compuware-Corporation, SUN-Microsystems: Xmof queries, views and transformations on models using mof, ocl and patterns. OMG, OMG Document ad/2003-08-07 (August 2003)
23. OMG: Mof 2.0 query / views / transformations rfp (April 2004)
24. Lengyel, L., Levendovszky, T., Charaf, H.: Constraint Validation Support in Visual Model Transformation Systems. Acta Cybernetica 17(2), 339–357 (2005)
25. White, J., Gray, J., Schmidt, D.C.: Constraint-based model weaving. In: Transactions on Aspect-Oriented Software Development (2009) (to appear)
26. Gajos, K., Weld, D.S.: Supple: automatically generating user interfaces. In: IUI 2004: Proceedings of the 9th international conference on Intelligent user interfaces, pp. 93–100. ACM Press, New York (2004)
27. Palanque, P., Paterno, F.: Formal Methods in Human-Computer Interaction. Springer, Berlin (1998) ISBN 978-3540761587
28. Browne, T., Davila, D., Rugaber, S., Stirewalt, R.E.K.: The mastermind user interface generation project. Technical report, Georgia Institute of Technology (1996)
29. Hettel, T., Lawley, M., Raymond, K.: Model synchronisation: Definitions for round-trip engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)
30. Cicchetti, A., Di Ruscio, D., Eramo, R.: Towards propagation of changes by model approximations. In: EDOCW 2006: Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops, Washington, DC, USA, p. 24. IEEE Computer Society, Los Alamitos (2006)

Automatic Model Generation Strategies for Model Transformation Testing

Sagar Sen, Benoit Baudry, and Jean-Marie Mottu

INRIA, Centre Rennes - Bretagne Atlantique, Campus universitaire de beaulieu, Rennes Cedex 35000, France
{ssen,bbaudry,jmottu}@irisa.fr

Abstract. Testing *model transformations* requires input models which are graphs of inter-connected objects that must conform to a meta-model and meta-constraints from heterogeneous sources such as well-formedness rules, transformation pre-conditions, and test strategies. Manually specifying such models is tedious since models must simultaneously conform to several meta-constraints. We propose automatic model generation via constraint satisfaction using our tool Cartier for model transformation testing. Due to the virtually infinite number of models in the input domain we compare strategies based on input domain partitioning to guide model generation. We qualify the effectiveness of these strategies by performing mutation analysis on the transformation using generated sets of models. The test sets obtained using partitioning strategies gives mutation scores of up to 87% vs. 72% in the case of unguided/random generation. These scores are based on analysis of 360 automatically generated test models for the representative transformation of UML class diagram models to RDBMS models.

1 Introduction

Model transformations are core MDE components that automate important steps in software development such as refinement of an input model, re-factoring to improve maintainability or readability of the input model, aspect weaving, exogenous and endogenous transformations of models, and generation of code from models. Although there is wide spread development of model transformations in academia and industry there is mild progress in the domain of validating transformations. In this paper, we address the challenges in validating model transformations via *black-box testing*. We think that black-box testing is an effective approach to validating transformations due to the diversity of transformation languages based on graph rewriting [1], imperative execution (Kermeta [2]), and rule-based transformation (ATL [3]) that render language specific formal methods and white-box testing impractical.

In black-box testing of model transformations we require *test models* that can detect bugs in the model transformation. These models are graphs of inter-connected objects that must conform to a meta-model and satisfy meta-constraints such as well-formedness rules and transformation pre-conditions.

Automatic model generation based on constraint satisfaction is one approach to ensure that meta-constraints and test requirements are simultaneously satisfied by models. In previous work [4], we introduce a tool Cartier that transforms the input meta-model (in Eclipse Model Framework (EMF) standard [5]) of a model transformation and pre-condition (in a textual language such as OCL [6]) to a common constraint language Alloy. Cartier invokes Alloy to generate a Boolean CNF formula and solve it using a SAT solver [7] to obtain solutions at a low-level of abstraction. Cartier transforms these solutions back to instances of the high-level input meta-model (as XMI instances of the EMF meta-model). However, most models generated using Cartier are only trivial as they are not guided by a strategy. One of the goals of our work is to compare transformation independent strategies to guide automatic test model generation in order to detect bugs.

In this paper we use Cartier to systematically generate finite sets of models from the space of virtually infinite input models using different test strategies. First, we generate sets of random models (we call this strategy random for convenience although random or pseudorandom models is still not a well-defined concept and is not the focus of this paper). We also generate sets of models guided by model fragments obtained from meta-model partitioning strategies presented in our previous work [8]. We compare strategies using mutation analysis of model transformations [9] [10]. Mutation analysis serves as a *test oracle* to determine the relatively adequacy of generated test sets. We do not use domain-specific post-conditions as oracles to determine the correctness of the output models. We use the representative model transformation of Unified Modelling Language Class Diagram (UMLCD) to Relational Database Management Systems (RDBMS) models called class2rdbms to illustrate our model generation approach and effectiveness of test strategies. The mutation scores show that input domain partitioning strategies guide model generation with considerably higher bug detection abilities (87%) compared to unguided generation (72%). Our results are based on 360 generated test models and about 50 hours of computation on a high-end server. We summarize our contributions as follows:

– **Contribution 1:** We use Cartier [4] to generate hundreds of valid models using different search strategies.
– **Contribution 2:** We use mutation analysis [10] to compare sets of generated models for their bug detecting effectiveness. We show that model sets generated using partitioning strategies, previously presented in our work [8], help detect more bugs than unguided generation.

The paper is organized as follows. In Section 2 we present the transformation testing problem and the case study. In Section 3 we describe our tool Cartier for automatic model generation, strategies for guiding model generation, and mutation analysis for model transformation testing. In Section 5 we present the experimental methodology, setup and results to compare model generation strategies. In Section 6 we present related work. We conclude in Section 7.

2 Problem Description

We present the problem of black-box testing *model transformations*. A model transformation $MT(I,O)$ is a program applied on a set of input models I to produce a set of output models O as illustrated in Figure 1. The set of all input models is specified by a meta-model MM_I (UMLCD in Figure 2). The set of all output models is specified

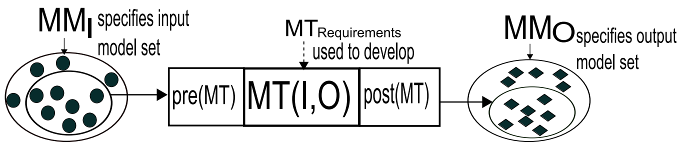

MMI specifies input model set
MTRequirements used to develop
pre(MT)
MT(I,O)
post(MT)
MMO specifies output model set

Fig. 1. A Model Transformation

by meta-model MM_O. The pre-condition of the model transformation $pre(MT)$ further constrains the input domain. A post-condition $post(MT)$ limits the model transformation to producing a subset of all possible output models. The model transformation is developed based on a set of requirements $MT_{Requirements}$.

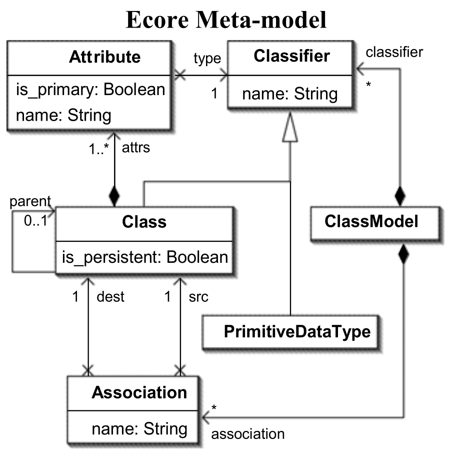
Model generation for black-box testing involves finding valid input models we call *test models* from the set of all input models I. Test models must satisfy constraints that increase the trust in the quality of these models as test data and thus should increase their capabilities to detect bugs in the model transformation $MT(I,O)$. Bugs may also exist in the input meta-model and its invariants MM_I or the transformation pre-condition $pre(MT)$. However, in this paper we only focus on detecting bugs in a transformation.

2.1 Transformation Case Study

Our case study is the transformation from UML Class Diagram models to RDBMS models called class2rdbms. In this section we briefly describe class2rdbms and discuss why it is a representative transformation to validate test model generation strategies.

In black-box testing we need input models that conform to the input meta-model MM_I and transformation pre-condition $pre(MT)$. Therefore, we only discuss the MM_I and $pre(MT)$ for class2rdbms and avoid discussion of the model transformation output domain. In Figure 2 we present the input meta-model for class2rdbms. The concepts and relationships in the input meta-model are stored as an Ecore model [5] (Figure 2 (a)). The invariants on the UMLCD Ecore model, expressed in Object Constraint Language (OCL) [6], are shown in Figure 2 (b). The Ecore model and the invariants together represent the input meta-model for class2rdbms. The OCL and Ecore are industry standards used to develop meta-models and specify different invariants on them. OCL is not a domain-specific language to specify invariants. However, it is designed to formally encode natural language requirements specifications independent of its domain. In [11] the authors present some limitations of OCL.

The input meta-model MM_I gives an initial specification of the input domain. However, the model transformation itself has a pre-condition $pre(MT)$ that test models need to satisfy to be correctly processed. Constraints in the pre-condition for class2rdbms include: (a) All Class objects must have at least one primary Attribute object (b) The type of an Attribute object can be a Class C, but finally the transitive closure of the type of Attribute objects of Class C must end with type PrimitiveDataType. In our case we approximate this recursive closure constraint by stating that Attribute object can be of type Class up to a depth of 3 and the 4th time it should have a type PrimitiveDataType. This is a finitization operation to avoid navigation in an infinite loop. (c) A Class object



(a)

OCL Invariants

```
context Class

    inv noCyclicInheritance:
        not self.allParents()->includes(self)

    inv uniqueAttributesName:
        self.attrs->forAll(att1, att2 |
            att1.name=att2.name implies att1=att2)

context ClassModel

    inv uniqueClassifierNames:
        self.classifier->forAll(c1, c2 |
            c1.name=c2.name implies c1=c2)

    inv uniqueClassAssociationSourceName :
        self.association->forAll(ass1, ass2 |
            ass1.name=ass2.name implies
            (ass1=ass2 or ass1.src != ass2.src))
```

(b)

Fig. 2. (a) Simple UML Class Diagram Ecore Model (b) OCL constraints on the Ecore model

cannot have an Association and an Attribute object of the same name (d) There are no cycles between non-persistent Class objects.

We choose class2rdbms as our representative case study to validate input selection strategies. It serves as a sufficient case study for several reasons. The transformation is the benchmark proposed in the MTIP workshop at the MoDELS 2005 conference [12] to experiment and validate model transformation language features. The input domain meta-model of UML class diagram model covers all major meta-modelling concepts such as inheritance, composition, finite and infinite multiplicities. The constraints on the UML meta-model contain both first-order and higher-order constraints. There also exists a constraint to test transitive closure properties on the input model such as there must be no cyclic inheritance. The class2rdbms exercises most major model transformation operators such as navigation, creation, and filtering (described in more detail in [10]) enabling us to test essential model transformation features. Among the limitations the simple version of the UMLCD meta-model does not contain Integer and Float attributes. The number of classes in the simplified UMLCD meta-model is not very high when compared to the standard UML 2.0 specification. There are also no inter meta-model references and arbitrary containments in the simple meta-model.

Model generation is relatively fast but performing mutation analysis is extremly time consuming. Therefore, we perform mutation analysis on class2rdbms to qualify transformation and meta-model independent strategies for model synthesis. If these strategies prove to be useful in the case of class2rdbms then we recommend the use of these strategies to guide model synthesis in the input domain of other model transformations as an initial test generation step. For instance, in our experiments, we see that generation of a 15 class UMLCD models takes about 20 seconds and mutation analysis of a set of 20 such models takes about 3 hours on a multi-core high-end server. Generating thousands of models for different transformations takes about 10% of the time while performing mutation analysis takes most of the time.

3 Automatic Model Generation

We use the tool Cartier previously introduced in our paper [4] to automatically generate models. We invoke Cartier to transform the input domain specification of a model transformation to a common constraint language Alloy. Then Cartier invokes the Alloy API to obtain Boolean CNF formulae [13], launch a SAT solver such as ZChaff [7] to generate models that conform to the input domain of a model transformation.

Cartier transforms a model transformation's input meta-model expressed in the Eclipse Modelling Framework [5] format called Ecore using the transformation rules presented in [4]. OCL constraints and natural language constraints on the input ecore meta-model are manually transformed to Alloy facts. These OCL constraints are used to express meta-model invariants and model transformation pre-conditions. We do not automate OCL to Alloy as there are several challenges posed by this transformation as discussed in [14]. We do not claim that all OCL constraints can be manually/automatically transformed to Alloy for our approach to be applicable in the most general case. OCL and Alloy were designed with different goals. OCL is used mainly to query a model and check if certain invariants are satisfied. Alloy facts and predicates on the other hand enforce constraints on a model. This is in contrast with the side-effect free OCL. The core of Alloy is declarative and is based on first-order relational logic with quantifiers while OCL includes higher-order logic and has imperative constructs to call operations and messages making some parts of OCL more expressive. In our case study, we have been successful in transforming all meta-constraints on the UMLCD meta-model to Alloy from their original OCL specifications. Identifying a subset of OCL that can be automatically transformed to Alloy is an *open challenge*. As an example transformation consider the invariant for no cyclic inheritance in Figure 2(b). The constraint is specified as the following fact:

```
fact noCyclicInheritance {no c: Class | c in c.^parent}
```

The generated Alloy model for the the UMLCD meta-model is given in Appendix A. This Alloy model only describes the input domain of the transformation. Solving the facts and signatures in the model (see Section 3.2) results in *unguided and trivial solutions*. Are these trivial solution capable of detecting bugs? This is the question that is answered in Section 5. Are there better heuristics to generate test models? In the following sub-section we illustrate how one can guide model generation using strategies based on input domain partitioning.

3.1 Strategies to Guide Model Generation

Good strategies to guide automatic model generation are required to obtain test models that detect bugs in a model transformation. We define a strategy as a process that generates *Alloy predicates* which are constraints added to the Alloy model synthesized by Cartier as described in Section 3. This combined Alloy model is solved and the solutions are transformed to model instances of the input meta-model that satisfy the predicate. We present the following strategies to guide model generation:

- **Random/Unguided Strategy:** The basic form of model generation is unguided where only the Alloy model obtained from the meta-model and transformation is used to generate models. No extra knowledge is supplied to the solver in order to generate models. The strategy yields an empty Alloy predicate *pred random* {}.
- **Input-domain Partition based Strategies:** We guide generation of models using test criteria to combine *partitions* on domains of all properties of a meta-model (cardinality of references or domain of primitive types for attributes). A *partition* of a set of elements is a collection of n ranges A_1,..., A_n such that A_1, ..., A_n do not overlap and the union of all subsets forms the initial set. These subsets are called *ranges*. We use partitions of the input domain since the number of models in the domain are infinitely many. Using partitions of the properties of a meta-model we define two test criteria that are based on different strategies for combining partitions of properties. Each criterion defines a set of *model fragments* for an input meta-model. These fragments are transformed to predicates on meta-model properties by Cartier. For a set of test models to cover the input domain at least one model in the set must cover each of these model fragments. We generate model fragment predicates using the following test criteria to combine partitions (cartesian product of partitions):
    - **AllRanges Criteria:** AllRanges specifies that each range in the partition of each property must be covered by at least one test model.
    - **AllPartitions Criteria:** AllPartitions specifies that the whole partition of each property must be covered by at least one test model.

The notion of test criteria to generate model fragments was initially proposed in our paper [8]. The accompanying tool called Meta-model Coverage Checker (MMCC) [8] generates model fragments using different test criteria taking any meta-model as input. Then, the tool automatically computes the coverage of a set of test models according to the generated model fragments. If some fragments are not covered, then the set of test models should be improved in order to reach a better coverage.

In this paper, we use the model fragments generated by MMCC for the UMLCD Ecore model (Figure 2). We use the criteria AllRanges and AllPartitions. For example, in Table 1, *mfAllRanges1* and *mfAllRanges2* are model fragments generated by Cartier using MMCC [8] for the *name* property of a classifier object. The *mfAllRanges1* states that there must be at least one classifier object with an empty name while *mfAllRanges2* states that there must be at least one classifier object with a non-empty name. These values for name are the ranges for the property. The model fragments chosen using AllRanges *mfAllRanges1* and *mfAllRanges2* define two partitions *partition1* and *partition2*. The model fragment *mfAllPartitions1* chosen using AllPartitions defines both *partition1* and *partition2*.

These model fragments are transformed to Alloy predicates by Cartier. For instance, model fragment mfAllRanges7 is transformed to the predicate :

```
pred mfAllRanges7(){some c:Class|#c.attrs=1}
```

As mentioned in our previous paper [8] if a test set contains models where all model fragments are contained in at least one model then we say that the input domain is completely covered. However, these model fragments are generated considering only the concepts and relationships in the Ecore model and they do not take into account the constraints on the Ecore model. Therefore, not all model fragments are consistent with the input meta-model because the generated models that contain these model fragments do not satisfy the constraints on the meta-model. Cartier invokes the Alloy Analyzer [15] to automatically check if a model containing a model fragment and satisfying the input domain can be synthesized for a general scope of number of objects. This allows us to *detect inconsistent model fragments*. For example, the following predicate, mfAllRanges7a, is the Alloy representation of a model fragment specifying that some Class object does not have any Attribute object. Cartier calls the Alloy API to execute the run statement for the predicate mfAllRanges7a along with the base Alloy model to create a model that contains up to 30 objects per class/concept/signature:

```
pred mfAllRange7a(){some c:Class|#c.attrs=0}
run mfAllRanges7 for 30
```

The Alloy analyzer yields a *no solution* to the run statement indicating that the model fragment is not consistent with the input domain specification. This is because no model can be created with this model fragment that also satisfies an input domain constraint that states that every Class must have at least one Attribute object:

```
sig Class extends Classifier{..attrs : some Attribute..}
```

, where *some* indicates 1..*. However, if a model solution can be found using Alloy we call it a *consistent model fragment*. MMCC generates a total of 15 consistent model fragments using AllRanges and 5 model fragments using the AllPartitions strategy, as shown in Table 1.

Table 1. Consistent Model Fragments Generated using AllRanges and AllPartitions Strategies

Model-Fragment	**Description**
mfAllRanges1	A Classifier $c \mid c.name =$“”
mfAllRanges2	A Classifier $c \mid c.name! =$“”
mfAllRanges3	A Class $c \mid c.is\_persistent = True$
mfAllRanges4	A Class $c \mid c.is\_persistent = False$
mfAllRanges5	A Class $c \mid \#c.parent = 0$
mfAllRanges6	A Class $c \mid \#c.parent = 1$
mfAllRanges7	A Class $c \mid \#c.attrs = 1$
mfAllRanges8	A Class $c \mid \#c.attrs > 1$
mfAllRanges9	An Attribute $a \mid a.is\_primary = True$
mfAllRanges10	An Attribute $a \mid a.name =$“”
mfAllRanges11	An Attribute $a \mid a.name! =$“”
mfAllRanges12	An Attribute $a \mid \#a.type = 1$
mfAllRanges13	An Association $as \mid as.name =$“”
mfAllRanges14	An Association $as \mid \#as.dest = 0$
mfAllRanges15	An Association $as \mid \#as.dest = 1$
mfAllPartitions1	Classifiers $c1, c2 \mid c1.name =$“” and $c2.name! =$“”
mfAllPartitions2	Classes $c1, c2 \mid c1.is\_persistent = True$ and $c2.is\_persistent = False$
mfAllPartitions3	Classes $c1, c2 \mid \#c1.parent = 0$ and $\#c2.parent = 1$
mfAllPartitions4	Attributes $a1, a2 \mid a1.is\_primary = True$ and $a2.is\_primary = False$
mfAllPartitions5	Associations $as1, as2 \mid as1.name =$“” and $as2.name! =$“”

3.2 Model Generation by Solving Alloy Model

Given the base Alloy model with signatures, facts and predicates from model fragments (see A) Cartier synthesizes Alloy run commands. Cartier synthesizes a run command for a given scope or based on exact number of objects per class/signature. A scope is the maximum number of objects/atoms per signature. Scope can be specified for individual signatures or the same scope can apply to all signatures. Executing the following run command in Alloy attempts to generates a model that conforms to the input domain and satisfies the model fragment called mfAllRanges1.

```
run mfAllRanges1 for 20
```

Cartier invokes a SAT solver using the Alloy API to incrementally increase the scope unto 20 and see if one or more solutions can be found. If solutions can be found we transform the low-level Alloy XML output to XMI that can be read by Ecore based model transformations or editors. On the other hand, we can also specify the correct number of atoms/objects per signature as shown below.

```
run mfAllRanges1 for for 1 ClassModel,5 int, exactly 5 Class,
exactly  25 Attribute, exactly 4 PrimitiveDataType,exactly 5 Association
```

4 Qualifying Models: Mutation Analysis for Model Transformation Testing

We generate sets of test models using different strategies and qualify these sets via mutation analysis [9]. Mutation analysis involves creating a set of faulty versions or *mutants* of a program. A test set must distinguish the program output from all the output of its mutants. In practice, faults are modelled as a set of mutation operators where each operator represents a class of faults. A mutation operator is applied to the program under test to create each mutant. A mutant is killed when at least one test model detects the pre-injected fault. It is detected when program output and mutant output are different. A test set is relatively adequate if it kills all mutants of the original program. A mutation score is associated to the test set to measure its effectiveness in terms of percentage of the killed/revealed mutants.

We use the mutation analysis operators for model transformations presented in our previous work [10]. These mutation operators are based on three abstract operations linked to the basic treatments in a model transformation: the navigation of the models through the relations between the classes, the filtering of collections of objects, the creation and the modification of the elements of the output model. Using this basis we define several mutation operators that inject faults in model transformations:

Relation to the same class change (RSCC). The navigation of one association toward a class is replaced with the navigation of another association to the same class.

Relation to another class change (ROCC). The navigation of an association toward a class is replaced with the navigation of another association to another class.

Relation sequence modification with deletion (RSMD). This operator removes the last step off from a navigation which successively navigates several relations.

Relation sequence modification with addition (RSMA). This operator does the opposite of RSMD, adding the navigation of a relation to an existing navigation.

Collection filtering change with perturbation (CFCP). The filtering criterion, which could be on a property or the type of the classes filtered, is disturbed.

Collection filtering change with deletion (CFCD). This operator deletes a filter on a collection; the mutant operation returns the collection it was supposed to filter.

Collection filtering change with addition (CFCA). This operator does the opposite of CFCD. It uses a collection and processes an additional filtering on it.

Class compatible creation replacement (CCCR). The creation of an object is replaced by the creation of an instance of another class of the same inheritance tree.

Classes association creation deletion (CACD). This operator deletes the creation of an association between two instances.

Classes association creation addition (CACA). This operator adds a useless creation of a relation between two instances.

Using these operators, we produced two hundred mutants from the class2rdbms model transformation with the repartition indicated in Table 2.

Table 2. Repartition of the UMLCD2RDBMS mutants depending on the mutation operator applied

Mutation Operator	CFCA	CFCD	CFCP	CACD	CACA	RSMA	RSMD	ROCC	RSCC	Total
Number of Mutants	19	18	38	11	9	72	12	12	9	200

In general, not all mutants injected become faults as some of them are equivalent and can never be detected. The controlled experiments presented in this paper uses mutants presented in our previous work [10]. We have clearly identified faults and equivalent mutants to study the effect of our generated test models.

5 Empirical Comparison of Generation Strategies

5.1 Experimental Methodology

We illustrate the methodology to qualify test generation strategies in Figure 3. The methodology flows is: (1) The inputs to Cartier are an Ecore meta-model, Alloy facts on the Ecore model, Alloy predicates for transformation pre-condition and experimental design parameters (such as factor levels, discussed shortly) (2) Cartier generates an Alloy model from the Ecore using rules in [4]. The input facts and predicates are inserted into the Alloy model. MMCC uses the Ecore to generate model fragments which Cartier transforms to Alloy predicates which are inserted into to the Alloy model. Cartier uses experiment design parameters to generate run commands and inserts them into the Alloy model. These aspects are inserted in the sequence: signatures, facts, predicate, and run commands (3) Cartier invokes the Alloy API to execute each run command (4) Cartier invokes run commands that uses the KodKod engine [13] in Alloy to transform the Alloy

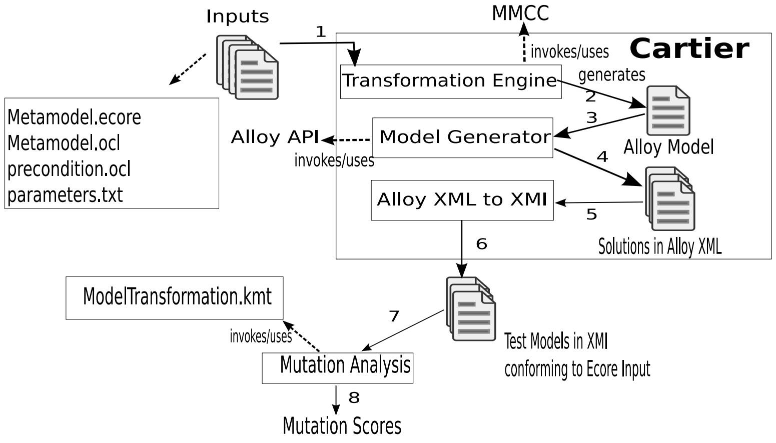


Fig. 3. Experimental Methodology to Qualify Automatic Model Generation Strategies

model to Boolean CNF, followed by invocation of a SAT solver such as ZChaff [7] to solve the CNF and generate solutions in Alloy XML (5,6) Cartier transforms Alloy XML instances to XMI using the input Ecore meta-model (7) We obtain XMI models in different sets for different strategies. Mutation analysis is performed on each of these sets with respect to a model transformation to give a mutation score for each set (8) We represent the mutation scores in a box-whisker diagram to compare and qualify strategies.

5.2 Experimental Setup and Execution

We use the methodology in Section 5.1 to compare model fragment driven test generation with unguided/random test model generation. We consider two test criteria for generating model fragments from the input meta-model: AllRanges and AllPartitions. We compare test sets generated using AllRanges and AllPartitions with randomly generated test sets containing an equal number of models. We use experimental design [16] to consider the effect of different factors involved in model generation. We consider the exact number of objects for each class in the input meta-model as factors for experimental design. The AllRanges criteria on the UMLCD meta-model gives 15 consistent model fragments (see Table 1). We have 15 models in a set, where each model satisfies one different model fragment. We synthesize 8 sets of 15 models using different levels for factors as shown in Table 3 (see rows 1,2,3,4,5,6). The total number of models in these 8 sets is 120. The AllPartitions criteria gives 5 consistent model fragments. We have 5 test models in a set, where each model satisfies a different model fragment. We synthesize 8 sets of 5 models using factor levels shown in Table 3. The levels for factors for AllRanges and AllPartitions are the same. Total number of models in the 8 sets is 40. The selection of these factors at the moment is not based on a problem-independent strategy. They are chosen based on the capacity of the solver in obtaining a model with 100 to 200 objects for our case study in a reasonable amount of time.

Table 3. Factors and their Levels for AllRanges and AllPartitions Test Sets

Factors	**Sets:** 1	2	3	4	5	6	7	8
#ClassModel	1	1	1	1	1	1	1	1
#Class	5	5	15	15	5	15	5	15
#Association	5	15	5	15	5	5	15	15
#Attribute	25	25	25	25	30	30	30	30
#PrimitiveDataType	4	4	4	4	4	4	4	4
Bit-width Integer	5	5	5	5	5	5	5	5
#Models/Set AllRanges	15	15	15	15	15	15	15	15
#Models/Set AllPartitions	5	5	5	5	5	5	5	

Table 4. Factor Levels for Random Pool of 200 Test Models

Factors	**Levels**
#ClassModel	1
#Class	5,10,15,20,25
#Association	5,10,15,20,25
#Attribute	25,30,35,40
#Primitive DataType	3,4
Bit-width Integer	5

Table 5. Mutation Scores in Percentage for All Test Model Sets

Set	1	2	3	4	5	6	7	8
Random **15 models/set in 8 sets**	72.6	70.61	69	71.64	72.68	72.16	69	69
AllRanges **15 models/set in 8 sets**	65.9	85.5	86.6	81.95	67.5	80.9	87.1	76.8
Random **5 models/set in 8 sets**	61.85	65.9	65.9	55.67	68.55	63.4	56.7	68.0
AllPartitions **5 models/set in 8 sets**	78.3	84.53	87.6	81.44	72.68	86.0	84	79.9

We create random/unguided models as a reference to qualify the efficiency of different strategies. We generate a pool of 200 unguided/random test models. We select this pool of test models using all the unique combinations of factor levels shown in the Table 4. We then randomly select 15 models at a time from this pool to create 8 sets of random models. We use these sets to compare mutation scores of 8 sets we obtain for the AllRanges strategy. Similarly, we randomly select 5 models at a time from the pool of 200 random models to create 8 sets of random models for comparison against the AllPartitions sets. The factor levels for random models as shown in Table 4. The levels range from very small to large levels covering a larger portion of the input domain in terms of model size allowing us to compare model fragments based test models against random test models of varying sizes.

To summarize, we generate 360 models using an Intel(R) Core^{TM} 2 Duo processor with 4GB of RAM. We perform mutation analysis of these sets to obtain mutation scores. The total computation time for the experiments which includes model generation and mutation analysis is about 50 hours. We discuss the results of mutation analysis in the following section.

5.3 Results and Discussion

Mutation scores for AllRanges test sets are shown in Table 5 (row 2). Mutation scores for test sets obtained using AllPartitions are shown in Table 5 (row 4). We discuss the effects of the influencing factors on the mutation score:

– The number of Class objects and Association objects has a strong correlation with the mutation score. There is an increase in mutation score with the level of these

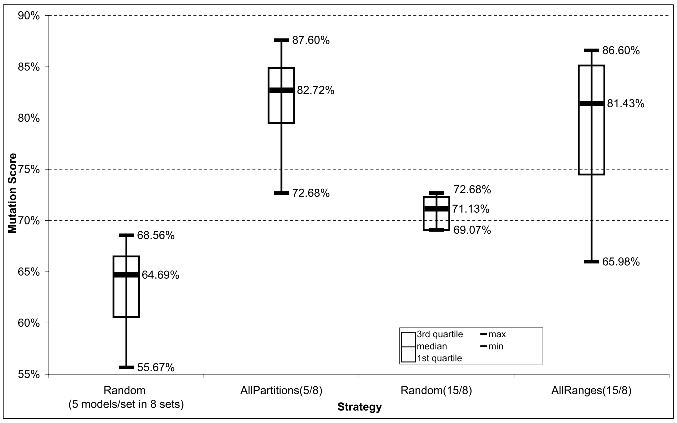


Fig. 4. Box-whisker Diagram to Compare Automatic Model Generation Strategies

factors. This is true for sets from random and model fragments based strategies. For instance, the lowest mutation score using AllRanges is 65.9 %. This corresponds to set 1 where the factor levels are 1,5,5,25,4,5 (see Column for set 1 in Table 3) and highest mutation scores are 86.6 and 87.1% where the factor levels are 1,15,5,25,4,5 and 1,5,15,25,4,5 respectively (see Columns for set 3 and set 7 in Table 3).

– We observe a strong correlation of the mutation score with the number of Class and Association objects due to the nature of the injected mutation operators. The creational, navigational, and filtering mutation operators injected in the model transformation are killed by input test models using a large number of Class and Association objects. However, we see that random models with both large and small number of Class and Association objects are not able to have a mutation score above 72%. There is a clear need for more knowledge to improve this mutation score.
– We observe that AllPartitions test sets containing only 5 models/set gives a score of maximum 87.1%. The AllPartitions strategy provides useful knowledge to improve efficiency of test models.

We random test sets with model fragment guided sets in the *box-whisker* diagram shown in Figure 4. The box whisker diagram is useful to visualize groups of numerical data such as mutation scores for test sets. Each box in the diagram is divided into lower quartile (25%), median, upper quartile (75% and above), and largest observation and contains statistically significant values. A box may also indicate which observations, if any, might be considered outliers or whiskers. In the box whisker diagram of Figure 4 we shown 4 boxes with whiskers for random sets and sets for AllRanges and AllPartitions. The X-axis of this plot represents the strategy used to select sets of test models and the Y-axis represents the mutation score for the sets.

We make the following observations from the box-whisker diagram:

– Both the boxes of AllRanges and AllPartitions represent mutation scores higher than corresponding random sets although the random sets were selected using models of larger size.

- The high median mutation scores for strategies AllRanges 81% and AllPartitions 82.7% indicate that both these strategies return consistently good test sets. The median for AllPartitions 82.72% is highest among all sets.
- The small size of the box for AllPartitions compared to the AllRanges box indicates its relative convergence to good sets of test models.
- The small set of 5 models/set using AllPartitions gives mutations scores equal or greater than 15 models/set using AllRanges. This implies that it is a more efficient strategy for test model selection. The main consequence is a reduced effort to write corresponding *test oracles* [17] with 5 models compared to 15 models.

The freely and automatically obtained knowledge from the input meta-model using the MMCC algorithm shows that AllRanges and AllPartitions are successful strategies to guide test generation. They have higher mutation scores with the same sources of knowledge used to generate random test sets. A manual analysis of the test models reveals that injection of inheritance via the parent relation in model fragments results in higher mutation scores. Most randomly generated models do not contain inheritance relationships as it is not imposed by the meta-model.

What about the 12% of the mutants that remain alive given that the highest mutation score is 87.6%? We note by an analysis of the live mutants that they are the same for both AllRanges and AllPartitions. There remain 25 live mutants in a total of 200 injected mutants (with 6 equivalent mutants). In the median case the AllRanges strategy gives a mutation score of 81.43% and while AllPartitions gives a mutation score of 82.73%. The live mutants in the median case are mutants not killed due to fewer objects in models. To consistently achieve a higher mutation score we need more CPU speed, memory and parallelization to efficiently generate large test models and perform mutation analysis on them. This extension of our work has not be been explored in the paper. It is important for us to remark that some live mutants can only be killed with more information about the model transformation such as those derived from its requirements specification. Further, not all model fragments are consistent with the input domain and hence they do not really cover the entire meta-model. Therefore, we miss killing some mutants. This information could help improve partitioning and combination strategies to generate better test sets.

We also neglect the effect of the constraint solver which is Alloy on the variation of the mutation scores. Relatively small boxes in the box-whisker diagram would be ideal to ascertain the benefits of test generation strategies. This again requires the generation of several thousand large and small models including multiple solutions for the same input specification. This will allow us to statistically minimize the external effects caused by Alloy and Boolean SAT solver allowing us to correctly qualify only the input generation strategies.

6 Related Work

We explore three main areas of related work : test criteria, automatic test generation, and qualification of strategies.

The first area we explore is work on test criteria in the context of model transformations in MDE. Random generation and input domain partitioning based test criteria are two widely studied and compared strategies in software engineering (non MDE) [18] [19] [20]. To extend such test criteria to MDE we have presented in [8] input domain partitioning of input meta-models in the form of model fragments. However, there exists no experimental or theoretical study to qualify the approach proposed in [8].

Experimental qualification of the test strategies require techniques for automatic model generation. Model generation is more general and complex than generating integers, floats, strings, lists, or other standard data structures such as dealt with in the Korat tool of Chandra et al. [21]. Korat is faster than Alloy in generating data structures such as binary trees, lists, and heap arrays from the Java Collections Framework but it does not consider the general case of models which are arbitrarily constrained graphs of objects. The constraints on models makes model generation a different problem than generating test suites for context-free grammar-based software [22] which do not contain domain-specific constraints.

Test models are complex graphs that must conform to an input meta-model specification, a transformation pre-condition and additional knowledge such as model fragments to help detect bugs. In [23] the authors present an automated generation technique for models that conform only to the class diagram of a meta-model specification. A similar methodology using graph transformation rules is presented in [24]. Generated models in both these approaches do not satisfy the constraints on the meta-model. In [25] we present a method to generate models given partial models by transforming the meta-model and partial model to a Constraint Logic Programming (CLP). We solve the resulting CLP to give model(s) that conform to the input domain. However, the approach does not add new objects to the model. We assume that the number and types of models in the partial model is sufficient for obtaining complete models. The constraints in this system are limited to first-order horn clause logic. In [4] we have introduce a tool Cartier based on the constraint solving system Alloy to resolve the issue of generating models such that constraints over both objects and properties are satisfied simultaneously. In this paper we use Cartier to systematically generate several hundred models driven by knowledge/constraints of model fragments [8]. Statistically relevant test model sets are generated from a factorial experimental design [16] [26].

The qualification of a set of test models can be based on several criteria such as code and rule coverage for white box testing, satisfaction of post-condition or mutation analysis for black/grey box testing. In this paper we are interested in obtaining the relative adequacy of a test set using mutation analysis [9]. In previous work [10] we extend mutation analysis to MDE by developing mutation operators for model transformation languages. We qualify our approach using a representative transformation UMLCD models to RDBMS models called class2rdbms implemented in the transformation language Kermeta [2]. This transformation [12] was proposed in the MTIP Workshop in MoDeLs 2005 as a comprehensive and representative case study to evaluate model transformation languages.

7 Conclusion

Black-box testing exhibits the challenging problem of developing efficient model generation strategies. In this paper we present Cartier, a tool to generate hundreds of models conforming to the input domain and guided by different strategies. We use these test sets to compare four strategies for model generation. All test sets using these strategies detect faults given by their mutation scores. We generate test sets using only the input meta-model. The comparison partitioning strategies with unguided generation taught us that both strategies AllPartitions and AllRanges look very promising. Partitioning strategies give a maximum mutation score of 87% compared to a maximum mutation score of 72% in the case of random test sets. We conclude from our experiments that the AllPartitions strategy is a promising strategy to consistently generate a small test of test models with a good mutation score. However, to improve efficiency of test sets we might require effort from the test designer to obtain test model knowledge/test strategy that take the internal model transformation design requirements into account.

References

1. Bardohl, R., Taentzer, G., Minas, M., Schurr, A.: Handbook of Graph Grammars and Computing by Graph transformation. In: vII: Applications, Languages and Tools. World Scientific, Singapore (1999)
2. Muller, P.A., Fleurey, F., Jezequel, J.M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
3. Jouault, F., Kurtev, I.: On the Architectural Alignment of ATL and QVT. In: Proceedings of ACM Symposium on Applied Computing (SAC 2006), FRA (April 2006)
4. Sen, S., Baudry, B., Mottu, J.M.: On combining multi-formalism knowledge to select test models for model transformation testing. In: IEEE International Conference on Software Testing, Lillehammer, Norway (April 2008)
5. Budinsky, F.: Eclipse Modeling Framework. The Eclipse Series. Addison-Wesley, Reading (2004)
6. OMG: The Object Constraint Language Specification 2.0, OMG: ad/03-01-07 (2007)
7. Mahajan, Y.S., Fu, Z., Malik, S.: Zchaff2004: An efficient sat solver. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 360–375. Springer, Heidelberg (2005)
8. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Towards dependable model transformations: Qualifying input test data. In: Software and Systems Modelling (2007) (accepted)
9. DeMillo, R., Lipton, R., Sayward, F.: Hints on test data selection: Help for the practicing programmer. IEEE Computer 11(4), 34–41 (1978)
10. Mottu, J.M., Baudry, B., Traon, Y.L.: Mutation analysis testing for model transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 376–390. Springer, Heidelberg (2006)
11. Vaziri, M., Jackson, D.: Some shortcomings of ocl, the object constraint language of uml. In: TOOLS 2000: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00), Washington, DC, USA. IEEE Computer Society, Los Alamitos (2000)
12. Bezivin, J., Rumpe, B., Schurr, A., Tratt, L.: Model transformations in practice workshop. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 120–127. Springer, Heidelberg (2006)
13. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Tools and Algorithms for Construction and Analysis of Systems, Braga, Portugal (March 2007)
14. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)

15. Jackson, D. (2008), http://alloy.mit.edu
16. Pfleeger, S.L.: Experimental design and analysis in software engineering. Annals of Software Engineering, 219–253 (2005)
17. Mottu, J.M., Baudry, B., Traon, Y.L.: Model transformation testing: Oracle issue. In: Proc. of MoDeVVa workshop colocated with ICST 2008, Lillehammer, Norway (April 2008)
18. Vagoun, T.: Input domain partitioning in software testing. In: HICSS 1996: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS), Washington, DC, USA. Decision Support and Knowledge-Based Systems, vol. 2 (1996)
19. Weyuker, E.J., Weiss, S.N., Hamlet, D.: Comparison of program testing strategies. In: TAV4: Proceedings of the symposium on Testing, analysis and verification, pp. 1–10. ACM, New York (1991)
20. Gutjahr, W.J.: Partition testing versus random testing: the influence of uncertainty. IEEE TSE 25, 661–674 (1999)
21. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on java predicates. In: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (2002)
22. Hennessy, M., Power, J.F.: An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In: Proc. of the 20th IEEE/ACM ASE, New York, USA (2005)
23. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Proceedings of ISSRE 2006, Raleigh, NC, USA (2006)
24. Ehrig, K., Kster, J., Taentzer, G., Winkelmann, J.: Generating instance models from meta models. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 156–170. Springer, Heidelberg (2006)
25. Sen, S., Baudry, B., Precup, D.: Partial model completion in model driven engineering using constraint logic programming. In: International Conference on the Applications of Declarative Programming (2007)
26. Federer, W.T.: Experimental Design: Theory and Applications. Macmillan, Basingstoke (1955)

A Concise Version of Alloy Model Synthesized by Cartier

```
module tmp/simpleUMLCD
open util/boolean as Bool
sig ClassModel{classifier:set Classifier,association:set Association}
abstract sig Classifier{name :  Int}
sig PrimitiveDataType extends Classifier {}
sig Class extends Classifier{
is_persistent: one Bool,parent : lone Class,attrs : some Attribute}
sig Association{name: Int,dest: one Class,src: one Class}
sig Attribute{name: Int,is_primary : Bool,type: one Classifier}
//Meta-model constraints
//There must be no cyclic inheritance in the generated class diagram
fact noCyclicInheritance {no c: Class|c in c.^parent}
/*All the attributes in a Class must have unique attribute names*/
fact uniqueAttributeNames {
all c:Class|all a1:  c.attrs, a2: c.attrs |a1.name==a2.name=>a1=a2}
//An attribute object can be contained by only one class
```

```
fact attributeContainment {
all c1:Class, c2:Class | all a1:c1.attrs, a2:c2.attrs|a1==a2=>c1=c2}
//There is exactly one ClassModel object
fact oneClassModel {#ClassModel=1}
/*All Classifier objects are contained in a ClassModel*/
fact classifierContainment {
all c:Classifier | c in ClassModel.classifier}
//All Association objects are contained in a ClassModel
fact associationContainment {
all a:Association| a in ClassModel.association}
/*A Classifier must have a unique name in the class diagram*/
fact uniqueClassifierName {
all c1:Classifier, c2:Classifier |c1.name==c2.name => c1=c2}
/*An associations have the same name either
they are the same association or they have different sources*/
fact uniqeNameAssocSrc {all a1:Association, a2:Association |
a1.name == a2.name => (a1 = a2 or a1.src != a2.src)}
/*Model Transformation Pre-condition*/
fact atleastOnePrimaryAttribute {
all c:Class| one a:c.attrs | a.is_primary==True}
fact no4CyclicClassAttribute{
all a:Attribute |a.type in Class => all a1:a.type.attrs|a1.type in
Class=>all a2:a.type.attrs|a2.type in Class=>all a3:a.type.attrs|a3.type
 in Class => all a4:a.type.attrs| a4.type in PrimitiveDataType}
fact noAttribAndAssocSameName{all c:Class,assoc:Association |
all a:c.attrs|(assoc.src==c)=>a.name!=assoc.name}
fact no1CycleNonPersistent {
all a: Association | (a.dest == a.src) => a.dest.is_persistent= True }
fact no2CycleNonPersistent{all a1: Association, a2:Association |
(a1.dest == a2.src and a2.dest==a1.src) =>
a1.src.is_persistent= True or a2.src.is_persistent=True}
```

A Simple Game-Theoretic Approach to Checkonly QVT Relations

Perdita Stevens

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract. The QVT Relations (QVT-R) transformation language allows the definition of bidirectional model transformations, which are required in cases where a two (or more) models must be kept consistent in the face of changes to either. A QVT-R transformation can be used either in checkonly mode, to determine whether a target model is consistent with a given source model, or in enforce mode, to change the target model. Although the most obvious semantic issues in the QVT standard concern the restoration of consistency, in fact even checkonly mode is not completely straightforward; this mode is the focus of this paper. We need to consider the overall structure of the transformation as given by when and where clauses, and the role of trace classes. In the standard, the semantics of QVT-R are given both directly, and by means of a translation to QVT Core, a language which is intended to be simpler. In this paper, we argue that there are irreconcilable differences between the intended semantics of QVT-R and those of QVT Core, so that the translation cannot be helpful. Treating QVT-R directly, we propose a simple game-theoretic semantics. We demonstrate that consistent models may not possess a single trace model whose objects can be read as traceability links in either direction. We briefly discuss the effect of variations in the rules of the game, to elucidate some design choices available to the designers of the QVT-R language.

1 Introduction

Model-driven development (MDD) is widely agreed to be an important ingredient in the development of reliable, maintainable multi-platform software. The Object Management Group, OMG, is the industry's consensus-based standards body, so the standards it proposes for model-driven development are necessarily important. In the area of MDD, a key standard is Queries, Views and Transformations (QVT, [5]), a specification of three different languages for defining *transformations* between models, which may include defining a restricted *view* of a model which abstracts away from aspects of the model not relevant to a particular class of intended user. Rather disappointingly, however, the Queries, Views and Transformations languages have been slow to be adopted. Few tools are available for any of the languages: notably, it sometimes happens that even those tools which use "QVT" in their marketing literature do not actually provide any of the three QVT languages, but rather, provide a "QVT-like" language. In this paper we will consider QVT Relations (QVT-R), the language which best permits the high-level, declarative specification of bidirectional transformations. There have been two candidate implementations of this: Medini QVT[1] and ModelMorf[2]. ModelMorf is the more faithful to [5], but unfortunately development of it seems to have ceased while the tool was still in pre-beta.

Why has the uptake of QVT been so low? Optimistically, we may point to the fact that, while the QVT standard has been under development for a long time, it has only recently been standardised. However, the same applied to other OMG standards, most notably UML, and did not prevent their adoption before finalisation. Lack of support for important engineering activities like testing and debugging may also play a role, but this does not explain why there *do* exist several tools each of which uses its own transformation language other than the OMG standard ones, and case studies of successful use of these tools. Perhaps a contributory factor is that, whereas the UML standard was developed following years of widespread use of various somewhat similar modelling languages, the model transformation arena is still far more sparsely populated. Therefore, how to define, or recognise, a good model transformation language for use on a particular problem is less well understood. We consider that the difficulty developers have in understanding the semantics of QVT may play a role, and we develop a game-theoretic semantics which we hope may be more accessible.

In this paper, we only consider transformations in checkonly mode. That is, we are interested in the case where a QVT-R transformation is presented with two models, and the transformation engine must return true if the models are consistent according to the definition of consistency embodied in the transformation, or false otherwise. Perhaps surprisingly, it turns out that this already raises some interesting issues.

Related work. This paper follows on from earlier work by the present author, [9], in which questions answered here, specifically the role of relation invocation in when and where clauses (relation definition applied to particular arguments), were left open. Discussion of the foundations of, and range of approaches to, bidirectionality, not specific to QVT, are presented in [8] and [7] respectively.

Greenyer and Kindler [3] presented at MODELS 2007 a discussion of the relationship between QVT Core and Triple Graph Grammars, together with an outline of a translation from QVT Core to TGGs. Romeikat and others [6] translated QVT-R transformations to QVT Operational. Garcia [1] formalised aspects of QVT-R in Alloy, permitting certain well-formedness errors to be detected.

Formal games have been widely used in computer science; the most relevant strand for this paper is surveyed in [10]. In modelling, the GUIDE tool [11] uses games to support design exploration and verification.

[1] http://projects.ikv.de/qvt/, version 1.6.0 current at time of writing.
[2] http://www.tcs-trddc.com/ModelMorf/index.htm, but download page not available at time of writing.

2 Background

QVT Relations. A QVT-R transformation is structured as a number of relations, connected by referencing one another in when and where clauses. The idea is that an individual relation constrains a tuple of models in a rather simple, local, way, by matching patterns rooted at model elements of particular kinds. The power, and the complexity, of the transformation comes from the way in which relations are connected. A relation may also have a when clause and/or a where clause. In these clauses, other relations are invoked with particular roots for their own patterns to be matched. In this way, global constraints on the models being compared can be constructed from a web of local constraints. The allowed dependencies between the choices made of values for variables – in a typical implementation, the order in which these choices are made – are such that the when functions as a kind of pre-condition; the where clause imposes further constraint on the values chosen during the relation to which it is attached (it is, in a way, a post-condition).

The reader is referred to [5] for details: the relevant sections are Chapter 7 and Appendix B. A key point is that the truth of a relation is defined using a logical formula which states that *for every* legal assignment of values to certain variables, *there must exist* an assignment of values to certain other variables, such that a given condition is satisfied.

Logic. In logical terms, this is expressed as a "for all–there exists" formula; more precisely, such a formula is called a Π_2 formula, provided that the formula which follows these two quantifiers is itself quantifier-free.

The difficulty in QVT-R is that actually, the truth of a complete transformation is expressed by a much more complex formula. Appendix B only expresses the truth of an individual relation, but this is defined in terms of the truth of the relations which may appear in its when and where clauses, so that, in fact, the number of alternations between universal and existential quantifiers (the length of a forall-thereexists-forall-thereexists... formula which would be equivalent to a whole QVT-R transformation evaluating to true) is unbounded. For example, consider the well-known example of transformation between UML class diagrams and RDBMS schemas, in which packages correspond to schemas, classes to tables and attributes to columns. Looking at [5] p197, we see that ClassToTable invokes relation AttributeToColumn in its where clause. The invocation gives explicit values for the root variables of the patterns in AttributeToColumn, but even though those are fixed, the usual rule applies as regards the rest of the valid bindings to be found in AttributeToColumn. Thus, for each valid binding of one pattern in ClassToTable (and of the when variables), there must exist a valid binding of the other pattern in ClassToTable, *such that* for each valid binding of the remaining variables of one pattern in AttributeToColumn (and of the when variables, except that in this case there are none), there exists a valid binding of the remaining variables of the other pattern in AttributeToColumn.[3] Note that, if there was more than one choice for

[3] Actually, the version in [5] is a little more complicated than this: AttributeToColumn invokes further relations in its where clause, and it is those which require the binding of remaining variables: but the point is the same.

the second binding in ClassToTable, it is entirely possible that it turns out that only one of these choices satisfies the rest of the condition, concerning the matching in AttributeToColumn: thus any evaluation, whether mental or by a tool, of ClassToTable has to be prepared either to consider both relations together, or to backtrack in the case that the first choice of binding made is not the best.

Therefore, while one might at first glance hope to be able to understand, and evaluate, the meaning of a QVT-R transformation by studying the relations individually, in fact, no such "local" evaluation is possible, because of the way the relations are connected.

Fortunately, similar situations arise throughout logic and computer science, and much work has been done on how to handle them. In particular, this is exactly the situation in which games have found to be a useful aid to developing intuition, as well as to formal reasoning.

Games. There is a long history in logic of formulating the truth of a logical proposition as the existence of a winning strategy in a two-player game. For example, the formula $\forall x.\exists y.y > x$ (where x and y are integers, say) can be turned into a game between two players. The player who is responsible for picking a value for x is variously called ∀belard, Player I, Spoiler, Refuter, depending on the community defining the game, while the player responsible for picking a value for y is called ∃louise, Player II, Duplicator or Verifier. We will go with Refuter and Verifier. Refuter's aim is, naturally, to refute the formula, while Verifier's aim is to verify it. In this game, Refuter has to pick a value for x, then Verifier has to pick a value for y. Verifier then wins this play of the game if $y > x$, while Refuter wins this play otherwise. In fact, in this case, Verifier has a *winning strategy* for the game: that is, she has a way of winning the game in the face of whatever moves Refuter may choose. This is an example of a two-player game of perfect information (that is, both players can see everything about one another's moves).

Of course, it is entirely possible that for a particular value of x, there is more than one value of y which makes the formula true: that is, Verifier has more than one winning strategy. When a Π_2 formula is true, a Skolem function expresses a particular set of choices that constitute a winning strategy: given x, it returns the chosen y. Different Skolem functions may exist which justify the truth of the same formula. In the example above, one choice of Skolem function maps x to $x + 1$, another maps x to $x+17$, another maps 1 to 23, 2 to 4, 3 also to 4, and so on. Clearly the trace model in QVT has something in common with a Skolem function.

Another family of examples comes from concurrency theory. Processes are modelled as labelled transition systems (LTSs), that is, an LTS is a set of states S including a distinguished start state $i \in S$, a set of labels L, and a ternary relation $\rightarrow \subseteq S \times L \times S$: we write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. The question of when two processes should be deemed to have consistent behaviour can be answered in many ways depending on context. One simple choice is *simulation.* A process $B = (S_B, i_B, L_B, \rightarrow_B)$ is said to simulate a process $A = (S_A, i_A, L_A, \rightarrow_A)$ if there exists a simulation relation $\mathcal{S} \subseteq S_A \times S_B$ containing (i_A, i_B). The condition for the relation to be a simulation relation is the following:

$$(s,t) \in \mathcal{S} \Rightarrow (\forall a, s' \,.(s \xrightarrow{a} s' \Rightarrow \exists t' \,.\, t \xrightarrow{a} t' \wedge (s', t') \in \mathcal{S}))$$

This can very easily be encoded as a game: starting at the start state of A, Refuter picks a transition. Verifier has to pick a transition from the start state of B which has the same label. We now have a new pair of states, the targets of the chosen transitions, and the process repeats: again, Refuter chooses a transition from A and Verifier has to match it. Play continues unless or until one player cannot go: either Refuter cannot choose a transition, because there are no transitions from his state, or Verifier cannot choose a transition because there is no transition from her state which matches the label on the transition chosen by Refuter. A player wins if the other player cannot move. If play continues for ever, Verifier wins. It is easy to show that in fact, Verifier has a winning strategy for this game exactly when there exists a simulation relation between the two processes; indeed, in a sense which can be made precise, a simulation relation *is* a winning strategy for Verifier. (As with the Skolem functions for Π_2 formulae, there may be more than one simulation relation between a given pair of processes.)

A curious and relevant fact about simulation is that even if B simulates A by simulation relation $\mathcal{S}$ *and* A simulates B by simulation relation $\mathcal{T}$, it does not follow that A simulates B by the reverse of $\mathcal{S}$, nor even that there must exist some relation which works as a simulation in both directions. This is the crucial difference between simulation equivalence and the stronger relation of bisimulation equivalence; see for example [2].

We will shortly define the semantics of QVT-R using a similar game, but first, we must consider an alternative approach.

3 The Translation from QVT Relations to QVT Core

In an attempt to help readers and connect the several languages it defines, [5] defines the semantics of QVT Relations both directly, and by translation to QVT Core. Both specifications are informal (notwithstanding some minor use of logic e.g. in Appendix B). [5] does not specify what should happen in the case of conflicts between the two, nor does it explicitly argue for their consistency. Therefore any serious attempt to provide a formally-based semantics for QVT-R needs to take both methods into consideration.

In this section, we consider the translation, with the aid of a very simple example QVT-R transformation. We then argue that, not only is what we believe to be the intended translation of this transformation not semantically equivalent, but also, the intended semantics of QVT Core appear to be such that it simply cannot express semantics equivalent to those of our simple QVT-R example. That is, even if our reading of the translation is incorrect, the problem remains: *no* translation can correctly reproduce the semantics of QVT-R. If the reader is convinced by the argument, it follows that the translation of QVT-R to QVT Core cannot contribute to an understanding of QVT-R.

Consider an extremely simple MOF metamodel which we will call SimplestMM. It defines one metaclass, called ModelElement, which is an instance of MOF's Class. It defines nothing else at all, so models which conform to this metamodel are simply collections (possibly empty) of instances of ModelElement. (Of course, in the usual object-oriented fashion, there is no obstacle to having several instances of ModelElement which are indistinguishable except by their identities.) We will refer to three models which conform to SimplestMM, having zero, one and two ModelElements respectively. We will imaginatively call them Zero, One and Two. Indeed, models conforming to SimplestMM can be identified in this way with natural numbers: a natural number completely determines such a model, and vice versa.

Next, consider a very simple QVT-R transformation between two models each of which conforms to SimplestMM. Figure 1 show the text of the transformation (we use ModelMorf syntax here).

```
transformation Translation (m1 : SimplestMM ; m2 : SimplestMM)
{
  top relation R
  {
    checkonly domain m1 me1:ModelElement {};
    checkonly domain m2 me2:ModelElement {};
  }
}
```

Fig. 1. A very simple transformation

Suppose that we use the QVT-R semantics to execute this transformation *in the direction of* `m2` (we will return to the issue of directionality of checkonly transformations below, in Section 4). When executed in the direction of `m2`, it should return true if and only if, for every valid binding of `me1` there exists a valid binding of `me2`. There are no constraints beyond the type specification, so this is equivalent to: if model `m1` is non-empty, then model `m2` must also be non-empty. If model `m1` is empty, then there is no constraint on model `m2`. Thus, when invoked on the six possible pairs of models from Zero, One and Two, the transformation should return false on the pairs (One,Zero) and (Two,Zero), otherwise true. Conversely, if we check in the direction of `m1`, the transformation returns false if `m1` is empty and `m2` is not, otherwise true. Reassuringly, ModelMorf gives exactly these results.

QVT-R works this way because its semantics are specified using logical "for all–there exists" formulae, without reference to a trace model or any other means of enforcing a permanent binding of one model element to another, such that a model element might be considered "used up". While [5] says that running a QVT-R transformation "implicitly" generates a trace model, the definition of the transformation does not rely upon its existence. It is simply assumed that an implementation will build a trace model, and use it, for example, to allow small changes to one model to be propagated to another without requiring all the computation involved in running a transformation to be redone. However, because the definition of QVT-R is independent of any trace model or its properties, there is no obstacle to the same model element being used more than once, which is why the transformation has the semantics discussed, rather than enforcing any more restrictive condition, such as that the two models have the same number of model elements. This helps to provide QVT-R the ability to express non-bijective transformations in the sense discussed in [9]; this ability in turn is essential to allow the expression of transformations between models which abstract away different things. The absolute requirement to be able to do this is most obvious when we consider a transformation between a fully-detailed model and an abstracted *view* onto it, where either the full model or the view may be updated (this is called the "view update problem" in databases). Even in transformations between models we might regard as equally detailed, though, it turns out that non-bijectiveness is essential. For example, in a realistic interpretation of a transformation between UML packages and RDBMS schemas, there are many schemas which are consistent with a given package, and many packages consistent with a given schema. See [9] for more discussion.

Now, taking [5] at face value, we expect to be able to translate this simple QVT-R transformation into a QVT Core transformation which has the same behaviour, and which, in particular, will return the same values when invoked on our simple models. The specification of the translation is not so clear that mistakes are impossible (e.g., possibly the multiple importing of the same metamodel is unnecessary), but this is what the author believes to be the intended translation:

```
module SimpleTransformation imports SimplestMM {
transformation Translation {
  m1 imports SimplestMM;
  m2 imports SimplestMM;
}

class TR {
  theM1element : ModelElement;
  theM2element : ModelElement;
}

map R in Translation {
check m1() {
  anM1element : ModelElement
}
check m2() {
  anM2element : ModelElement
}
where () {
  realize t:TR|
    t.theM1element = anM1element;
    t.theM2element = anM2element;
}
```

The effect of this QVT Core transformation is to construct for every model element in `m1` an object of the trace class `TR` which connects this model element to a corresponding model element in `m2`. However, [5] says several times that in QVT Core, valid bindings must be unique. For example, p133 says:

> There must be (exactly) one valid-binding of the bottom-middle pattern and (exactly) one valid binding of the bottom-domain pattern of a checked domain, for each valid combination of valid bindings of all bottom-domain-patterns of all domains not equal to the checked domain, and all these valid bindings must form a valid combination together with the valid bindings of all guard patterns of the mapping.

and this sentiment is then repeated in a logical notation. In executing the QVT Core version of our transformation on the models (Two,One), this condition would fail because, given the valid binding of the single ModelElement in One to variable `me2`, there would have to be two valid bindings to `me1`, one binding each of the ModelElements in Two. What is not so clear is whether this condition is intended to be satisfied if we run the example on (Two,Two): a literal reading would seen to suggest not, yet it seems impossible that QVT Core is intended to be unable to express the identity relation. The problem is *where* exactly the valid binding is supposed to be unique: in the model, or just in the mapping? That is, given a model element in m2, must there exist only one model element in m1 which could validly be linked to it, or is it, more plausibly, enough that there is only one model element which actually is linked to it by some trace object? Either way, though, (Two,One) will still fail.

Unfortunately no implementation of QVT Core seems to be available. Various sources refer to a pre-release of Compuware OptimalJ, but OptimalJ no longer exists. Therefore we cannot investigate what actual QVT Core tools do.

It is noteworthy, though, that this misapprehension that model elements, or at least patterns of them, must correspond one-to-one in order to make bidirectional transformations possible is pervasive: it appears even in the documentation for Medini QVT, which intends to be an implementation of QVT-R (see Medini QVT Guide, version 1.6, section QVT Relations Language, Bidirectionality).

Could we write a QVT Core transformation which did have the same behaviour as our simple QVT-R transformation? Unfortunately not. A moment's thought will show that the requirement that valid bindings correspond one-to-one (even if only in the constructed trace model) precludes any QVT Core transformation that could return true on both (One,Two) and (Two,One) but false on (One,Zero).

4 Transformation Direction

The reader who is familiar with [9] may have noticed an inconsistency between the treatment of bidirectional transformations in that paper and the way we described checkonly transformations above. The framework in [9] is based on a direction-free notion of consistency: a transformation between sets of models M and N specifies, for any pair $(m, n) \in M \times N$, whether or not m is consistent with n. In the above, however, our consistency check had a direction: checking

`Translation` in the direction of `m2` is not the same as checking it in the direction of `m1` and indeed, can give different answers. When `Translation` is checked in the direction of `m1` on the pair of models (Zero, One), it returns true, since there are no model elements on the left to be matched. When the same transformation is checked on the same pair of models in the other direction, it returns false.

The standard [5] is slightly ambivalent about whether a checkonly QVT-R transformation has a direction. Compare p13, which talks about "checking two models for consistency" and implicitly contrasts execution for enforcement, which has a direction, with execution for checking, which implicitly does not, with the details of the QVT-R definition which clearly assume that checking has a direction. The resolution seems to be (p19, my emphasis): A transformation can be executed in "checkonly" mode. In this mode, the transformation simply checks whether the relations hold **in all directions**, and reports errors when they do not.

That is, the notion of consistency intended by the QVT-R standard is given by conjunction: `m1` is consistent with `m2` according to transformation R if and only if R's check evaluates to true in both directions.

In fact, ModelMorf requires a transformation execution to have a direction specified, even when it is checkonly: to find out what the final result of a check-only transformation is, one has to manually run it in each direction and conjoin the results. Medini, by contrast, makes it impossible to run a transformation in checkonly mode: if you run a transformation in the direction of a domain which is marked enforce, there is no way to make the transformation engine return false if it finds that the models are inconsistent, rather than modifying the target model. These seems to be a misinterpretation of [5] and indeed is on the bug list. However, it is a superficial matter, because QVT-R is supposed to have "check then enforce" semantics: that is, it is not supposed to modify a model unless it is necessary to do so to enforce consistency. Therefore, given a QVT engine which was compliant with [5] except that it did not provide the ability to run transformations in checkonly mode, it would be easy to construct a fully compliant engine using a wrapper. The wrapper would save the target model, run the transformation, and compare the possibly modified target model with the original. If the target model had been modified, it would restore the original version and return false; otherwise, it would return true.

5 A Game-Theoretic Semantics for Checkonly QVT-R

Given a set of metamodels, a set of models conforming to the metamodels, a transformation written in a simplified version of QVT-R, and a direction for checking, we will define a formal game which explains the meaning of the transformation in the following sense. The game is played between Verifier and Refuter. Refuter's aim in the game is to refute the claim that the check should succeed; Verifier's aim is to verify that the check should succeed. The semantics of QVT is then defined by saying that the check returns true if and only if Verifier has a winning strategy for the game. If this is not the case, then (since by Martin's standard theorem on Borel determinacy [4] the game we will define will be determined, that is, one or other player will have a winning strategy) Refuter will have a winning strategy, and this corresponds to the check returning false.

This approach has several advantages. Most importantly, it separates out the specification of what the answer should be from the issue of how to calculate the answer efficiently. Calculating a winning strategy is often much harder (in both informal, and formal complexity, senses) than checking that a given strategy is in fact a winning strategy. Indeed, it can be useful to calculate a strategy using heuristics or other unsound or unproved methods, and then use a separate process to check that it is winning: this is the game equivalent of a common practice in formal proof, the separation between the simple process of proof checking and the arbitrarily hard process of proof finding. Nevertheless, although this paper does not address the issue of how winning strategies can be calculated efficiently, it is worth noting that formulating the problem in this way makes accessible a wealth of other work on efficient calculation of winning strategies to similar games.[4]

We may also hope to be able to use the game to explain the meaning of particular transformations, or of the QVT-R language in general, to developers or anyone else who needs to understand it: similar approaches have proven successful in teaching logic and concurrency theory.

Finally, a game-theoretic approach is a helpful framework in which to consider the implications of minor variations in decisions about what the meaning of a QVT-R transformation should be, since many such differences arise as minor variations in the rules of the game.

In order to specify a two-player game of perfect information, we need to provide definitions of the positions, the legal moves, the way to determine which player should move from a given position, and the circumstances under which each player shall win.

We fix a set of models, where each m_i conforms to a metamodel M_i, and a transformation definition given in a simplified version of QVT-R. Specifically, we consider that when and where clauses are only allowed to contain (conjunctions of lists of) relation invocations, not arbitrary OCL. We do not consider extension or overriding of transformations or relations. Further, our semantics is parametrised over a notion of pattern matching and relation-local constraint checking: in other words, we do not give semantics for these, but assume that an oracle is given to check the correctness, according to the relevant metamodel, of a player's allocation of values to variables, and local constraints such as identity of values between variables in different domains.

We will first define a game G_k which corresponds to the evaluation of a QVT-R checkonly transformation in the direction of one of its typed models, m_k. For ease of understanding we will explain the progress of the game informally first: Figure 2 defines the moves of the game more systematically. At every stage, if

[4] For the most complex games we consider here, such work is collated in the PGSolver project, http://www.tcs.ifi.lmu.de/pgsolver/. If we insist that the graph of relations should be a DAG, as discussed later in this section, simpler automata-based techniques suffice.

it is a player's turn to move, but that player has no legal moves available, then the other player wins.

To begin a play of game G_k, Refuter picks a top relation (call it R) and valid bindings for all patterns except that from m_k, and for any when variables (that is, variables which occur as arguments in relation invocations in the when clause of R). Notice that he is required to pick values which do indeed constitute valid bindings and satisfy relation-local constraints, as confirmed by the oracle mentioned earlier. Play moves to a position which we will notate $(\text{Verifier}, R, B, 1)$, indicating that Verifier is to move, that the relation in play is R, that bindings in set B have been fixed, and that only one of the players has yet played a part in this relation.

Verifier may now have a choice.

1. She may pick a valid binding for the as-yet-unbound variables from the m_k domain (if any), such that the relation-local constraints such as identity of values of particular variables are satisfied according the oracle. Let the complete set of bindings, including those chosen by both players, be B'. (If there are no more variables to bind, Verifier may still pick this and $B' = B$.) In this case, play moves to a position which we will notate $(\text{Refuter}, R, B', 2)$ indicating that Refuter is to move, that the relation in play is still R, that the bindings in set B' have been fixed, and that both players have now played their part in this relation.
2. Or, she can challenge one of the relation invocations in the when clause (if there are any), say S (whose arguments, note, have already been bound by Refuter). Then play moves to S, and before finishing her turn, she must pick valid bindings for all patterns of S except that from m_k, and for any when variables of S. Say that this gives a set of bindings C, in which the bindings of the root variables of all domains are those from B, and bindings of the other variables are those just chosen by Verifier. The new position is $(\text{Refuter}, S, C, 1)$.

If Verifier chose 2., play proceeds just as it did from $(\text{Verifier}, R, B, 1)$ *except that, notice, the roles of the players have been reversed.* It is now for Refuter to choose one of the two options above, in the new relation S.

If Verifier chose 1., Refuter's only option is to challenge one of the relation invocations in the where clause, say T (whose arguments, note, are bound). (If there are none, he has no valid move, and Verifier wins this play.) Then play moves to T, and, before finishing his turn, Refuter must pick valid bindings for all patterns of T except that from m_k, and for any when variables of T. Say that this gives a set of bindings D, in which the bindings of the root variables of all domains are those from B', and bindings of the other variables are those just chosen by Refuter. The new position is $(\text{Verifier}, T, D, 1)$. Play now continues just as above.

The final thing we have to settle is what happens if play never reaches a position where one of the players has no legal moves available: who wins an infinite play? We could just forbid this to happen, e.g., by insisting as a condition on QVT-R transformations that the graph in which nodes are relations and there

Position	Next position	Notes
Initial	$(\mathsf{Verif.}, R, B, 1)$	R is any top relation; B comprises valid bindings for all variables from domains other than k, and for any when variables.
$(P, R, B, 1)$	$(\overline{P}, R, B', 2)$	B' comprises B together with bindings for any remaining variables.
$(P, R, B, 1)$	$(\overline{P}, S, C, 1)$	S is any relation invocation from the when clause of R; C comprises B's bindings for the root variables of patterns in S, together with valid bindings for all variables from domains other than k in S, and for any when variables of S.
$(P, R, B, 2)$	$(\overline{P}, T, D, 1)$	T is any relation invocation from the where clause of R; D comprises B's bindings for the root variables of patterns in T, together with valid bindings for all variables from domains other than k in T, and for any when variables of T.

Fig. 2. Summary of the legal moves of the game G_k: note that the first element of the Position says who picks the next move, and that we write $\overline{P}$ for the player other than P, i.e. $\overline{\mathsf{Refuter}} = \mathsf{Verifier}$ and vice versa. Recall that bindings are always required to satisfy relevant metamodel and relation-local constraints.

is an edge from R to S if R invokes S in a where or when clause, should be acyclic. There is probably[5] a reasonable alternative that achieves sensible behaviour by allowing the winner of an infinite play to be determined by whether the outermost clause which is visited infinitely often is a where clause or a when clause: but this requires further investigation. Note that [5] has nothing to say about this situation: it corresponds to infinite regress of its definitions.

5.1 Discussion of the Treatment of When Clauses

Most of the above game definition is immediate from [5], but the treatment of when clauses requires discussion. From Chapter 7, ([5], p14): "The when clause specifies the conditions under which the relationship needs to hold, so the relation ClassToTable needs to hold only when the PackageToSchema relation holds between the package containing the class and the schema containing the table."

The naive way to interpret this would have been to say that both Refuter and Verifier choose their values, and then, if it turns out that the when clause is not satisfied given their choices, Verifier wins this play. This interpretation is not useful, however, as it often gives Verifier a way to construct a winning strategy which does not tell us anything interesting about the relationship between the models. When challenged by Refuter to pick a value for her domain, all she would need to do would be to pick a binding such that the when clause was not satisfied. In the case discussed by [5], whenever Refuter challenged with a class, she would reply with any table from a schema not corresponding to the package of his class, the when clause would not be satisfied, and she would win.

[5] By thinking from first principles about cases in which a play goes through a when (rsp. where) clause infinitely often, but only finitely often through where (rsp. when) clauses; or by intriguing analogy with μ calculus model-checking.

So the sense in which a when clause is a precondition must be more subtle than this. In programming, giving a function a precondition makes it easier for the function satisfy its specification, but here the idea is rather to restrict Verifier's choices: if Refuter chooses a class C in package P, Verifier is bound to reply not with any table, but specifically with a table T which is in a/the schema that corresponds to package P. The intuition behind allowing Verifier to counter-challenge the when clause is that Refuter may "unfairly" challenge Verifier to match the class from a/the "wrong" schema.

In trying to settle whether we really mean "a schema" or "the schema" in the paragraph above, we refer again to Appendix B of [5]. The problem is that this is not a complete definition. E.g., in order to use it to interpret ClassToTable, we already need to be able to determine whether, for given values of a package p and schema s, the when clause `when { PackageToSchema (p,s) }` holds. Informally it seems that people who write about QVT have two different interpretations of this, perhaps not always realising that they are different:

1. the purely relational: the pair `(p,s)` is any member of the relation expressed in `PackageToSchema`, *when it is interpreted using the very same text which we are now trying to interpret*
2. the operational: the program which is checking the transformation is assumed to have looked at `PackageToSchema` already and chosen a schema to correspond to package `p` (recording that choice using a trace object). According to this view, we only have to consider `(p,s)` if `s` is the very schema which was chosen on this run of the checking program.

To see the difference, imagine that there are two schemas, `s1` and `s2`, either of which could be chosen as a match for `p` in `PackageToSchema`. In the first interpretation, both possibilities have to be checked when `ClassToTable` is interpreted; in the second, only whichever one was actually used.

In our main game definition, we have taken the purely relational view, since we can do so while remaining compatible with the definitions in [5], whereas as we have seen in the SimplestMM example – which, recall, had no when or where clauses and whose semantics were therefore defined unambiguously by Appendix B – the idea that there should be a one-to-one correspondence between valid bindings is incompatible with Appendix B; but we will shortly consider a variant of the game which brings us closer to the latter view.

5.2 Variants of the Game

Non-directional variant. Let G be the variant of G_k in which, instead of a direction being defined as part of the game definition, Refuter is allowed to choose a direction ("once and for all") at the beginning of the play. Clearly, Verifier has a winning strategy for G if and only if she has a winning strategy for every G_k. This is the way of constructing a non-directional consistency definition from directional checks that is specified in [5]. However, note that it is not automatic that there should be any simple relationship between the various winning strategies; hence, there may not be any usable multi-directional trace relationship between the bindings in different models. Let us explain using an example derived from one in [2].

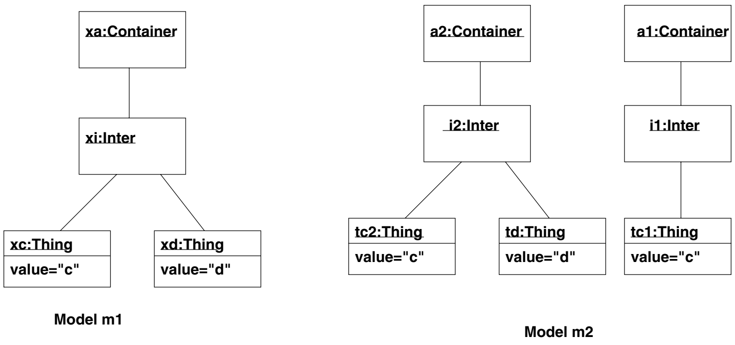


Fig. 3. m1 and m2 are (two-way) consistent according to QVT-R transformation Sim, but no set of bi-directional trace objects can link them

Figure 3 illustrates two models which conform to the obvious metamodel MM: a model may include multiple Containers, each of which references one Inter, each of which may reference multiple Things, each of which has a value. The following QVT-R transformation evaluates to true on the models shown, in both directions (both according to [5], and according to ModelMorf). Indeed, Verifier has a winning strategy for G: the only interesting choice she has to make is in G_2, where she has to be sure to reply with a2 (and i2), not a1 (and i1), if Refuter challenges in ContainersMatch by binding xa to c1 (and xi to inter1).

```
transformation Sim (m1 : MM ; m2 : MM)
{
  top relation ContainersMatch
  {
    inter1,inter2 : MM::Inter;
    checkonly domain m1 c1:Container {inter = inter1};
    checkonly domain m2 c2:Container {inter = inter2};
    where {IntersMatch (inter1,inter2);}
  }

  relation IntersMatch
  {
    thing1,thing2 : MM::Thing;
    checkonly domain m1 i1:Inter {thing = thing1};
    checkonly domain m2 i2:Inter {thing = thing2};
    where {ThingsMatch (thing1,thing2);}
  }

  relation ThingsMatch
  {
    s : String;
    checkonly domain m1 thing1:Thing {value = s};
    checkonly domain m2 thing2:Thing {value = s};
  }
}
```

Now, in the m1 direction the constructed trace will take a1 to xa, etc.; there is nothing else it can do. Yet in the m2 direction, a trace object which took xa to a1 would be erroneous. Thus *there can be no single set of trace objects whose links can be read in either direction, which could capture the correctness of this QVT-R transformation.*

Model-switching variant. Let G' be the variant of G in which, instead of the first player to move in a new relation being constrained to pick a valid binding everywhere except in the once-and-for-all designated target model m_k, the player is permitted to pick valid bindings for all but any one domain, making a new choice of which domain to leave out every time. This is a different way to define a non-directional variant of the game. The modification to the game rules is analogous to the difference, in concurrency theory, between a game which defines bisimulation equivalence and that which defines simulation equivalence. This might well have better properties as regards the existence of a sensible multi-directional trace model. This requires further investigation. Certainly in the example above, it will be Refuter who has a winning strategy for G': he will first challenge in m2 with a1, and later switch to m1 where he leads play to the "d" which cannot be matched starting from a1 in m2.

Trace-based variant. Let G^T be the variant of G in which, as play proceeds, we build a global auxiliary structure which records, for each relation, what choices of valid binding have been made by the players (for example, "Package P was matched with schema S"). It is an error if subsequent moves in a play try to choose differently (and we might consider a multi-directional subvariant in which *either* matching P with S' *or* matching S with P' was an error, along with uni-directional subvariants in which only one of those would be an error). The player to complete such an erroneous binding would immediately lose. Otherwise, play would be exactly as in G_k, *except that* it loops: if Refuter cannot go, he can "restart", choose a new top relation and play again, but the old auxiliary structure is retained. If play passes through infinitely many restarts, Verifier wins. This game would impose one-to-one constraints on valid bindings, and construct well-defined trace objects, at the expense of having a semantics incompatible with [5] and having curtailed expressivity.

6 Conclusions

We have presented a game-theoretic semantics of QVT-R checkonly transformations, based on the direct semantics in [5]; we justified our choice to ignore the translation to QVT Core by pointing out a fundamental incompatibility between the two languages. We have briefly discussed variants of the game, demonstrating in the process that bi-directional trace objects may not exist.

References

1. Garcia, M.: Formalization of QVT-Relations: OCL-based Static Semantics and Alloy-based Validation. In: Proceedings of the Second Workshop on MDSD Today, October 2008, pp. 21–30 (2008)

2. van Glabbeek, R.J.: The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, ch. 1, pp. 3–99. Elsevier, Amsterdam (2001)
3. Greenyer, J., Kindler, E.: Reconciling TGGs with QVT. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 16–30. Springer, Heidelberg (2007)
4. Martin, D.A.: Borel determinacy. Annals of Mathematics. Second series 102(2), 363–371 (1975)
5. OMG. MOF2.0 query/view/transformation (QVT) version 1.0. OMG document formal/2008-04-03 (2008), www.omg.org
6. Romeikat, R., Roser, S., Müllender, P., Bauer, B.: Translation of QVT relations into QVT operational mappings. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 137–151. Springer, Heidelberg (2008)
7. Stevens, P.: A landscape of bidirectional model transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
8. Stevens, P.: Towards an algebraic theory of bidirectional transformations. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 1–17. Springer, Heidelberg (2008)
9. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. Journal of Software and Systems Modeling, SoSyM (2009) (to appear)
10. Stirling, C.: Bisimulation, model checking and other games. In: Notes for Mathfit Instructural Meeting on Games and Computation (1997), http://homepages.inf.ed.ac.uk/cps/mathfit.ps
11. Tenzer, J., Stevens, P.: GUIDE: Games with UML for interactive design exploration. Journal of Knowledge Based Systems 20(7) (October 2007)

Supporting Model–Driven Development of Object–Relational Database Schemas: A Case Study

Juan Manuel Vara, Belén Vela, Verónica Andrea Bollati, and Esperanza Marcos

Kybele Research Group
Rey Juan Carlos University
Madrid, Spain
{juanmanuel.vara,belen.vela,veronica.bollati,
esperanza.marcos}@urjc.es

Abstract. This paper completes our proposal for automatic development of Object-Relational (OR) DataBase (DB) schemas. By means of a case study, this work focuses on presenting the tooling developed to support the whole process. As usual, the proposal starts from a conceptual data model (Platform Independent Model) depicted in a UML class diagram. Then, the conceptual data model is mapped into an OR DB model (Platform Specific Model) that represents the OR DB schema. To that end, we have implemented a set of formalized mapping rules using the ATL language. Finally, the SQL code that implements the modeled schema in Oracle 10*g* is automatically generated from the OR model by means of a MOFScript model to text transformation. Moreover, since the OR model could be refined along the design process, we have developed a graphical editor for OR DB models.

Keywords: Model-Driven Engineering, Object-Relational Databases, Model Transformations, Code Generation.

1 Introduction

In spite of the impact of relational Databases (DBs) over the last decades, this kind of DB has some limitations for supporting data persistence required by present day applications. Due to hardware improvements, more sophisticated applications have emerged, which can be characterized as consisting of complex objects related by complex relationships. Representing such objects and relationships in the relational model implies that the objects must be decomposed into a large number of tuples. Thus, a considerable number of joins are necessary to retrieve an object. Thus, when tables are too deeply nested, performance is significantly reduced [3]. A new DB generation appeared to solve these problems: the Object-Oriented DB generation, which includes Object–Relational (OR) DB [22]. This technology is well suited for storing and retrieving complex data. It supports complex data types and relationships, multimedia data, inheritance, etc. Moreover, it is based on a standard [10]. It extends the basic relational model with user defined types, collection types, typed tables, generalizations, etc. Nowadays, OR DBs are widely used both in the industrial and academic areas and they have been incorporated into a lot of commercial products.

Nevertheless, good technology is not enough to support these extensions. Just like it happens with relational DBs [8], design methodologies are needed for OR DB development. Those methodologies have to incorporate the OR model and to take into account old and new problems. Like choosing the right technology, solving platform migration and platform independence problems, maintenance, etc.

Following the Model-Driven Engineering (MDE) [5,21] proposal, this paper presents a model-driven approach for Object-Relational Databases development of MIDAS [13], a methodological framework for model-driven development of service-oriented applications. Since it is a model-driven proposal, models are considered as first class entities and mappings between the different models are defined, formalized and implemented. By means of a case study, this work presents the tooling developed to support the proposal.

The proposal starts from a Platform Independent Model (PIM), the conceptual data model, which is mapped to an OR DB model that represents the OR DB schema. To that end, we have defined metamodels for OR DB modeling. Then, the Platform Specific Model (PSM) is translated into SQL code by means of a model to text transformation. This work focuses on the mappings defined to transform the conceptual data model into the OR DB schema, i.e. to obtain the PSM from the PIM, as well as on the code generation from that schema.

The rest of the paper is structured as follows: first, section 2 summarizes the proposed OR DB development process, including the metamodel for OR DB modeling, as well as a brief overview of the PIM to PSM mappings. Next, section 3 uses a case study to illustrate the tooling developed to support the proposal. Section 4 examines related work and finally, section 5 outlines the main findings of the paper and raises a number of questions for future research.

2 Object-Relational DB Development in MIDAS

Our approach for OR DB development is framed in MIDAS, a model-driven methodology for service-oriented applications development. It falls on the proposal for the content aspect, which corresponds to the traditional concept of a DB. Here we focus on the PIM and PSM levels, whose models are depicted in Fig. 1. In our approach, the proposed data PIM is the conceptual data model represented with a UML class diagram while the data PSM is the OR model or the XML Schema model, depending on the selected technology to deploy the DB.

Regarding OR technology we consider two different platforms, thus two different PSMs. The first one is based on the SQL standard [10] and the second one on a specific product, Oracle10*g* [20]. Here, we focus on the OR DB development. We have defined two metamodels for OR DB modeling, one for the SQL standard and the other one for Oracle 10*g*. The later will be presented in the next section since the one for Oracle is still valid for SQL just by considering some minor differences. Moreover, using the one for Oracle we can check the output by loading the generated code into an Oracle DB.

In the proposed development process once the conceptual data model (PIM) is defined, an ATL model transformation generates the OR DB model (PSM). If needed, the designer could refine or modify this model using the graphical editor developed to

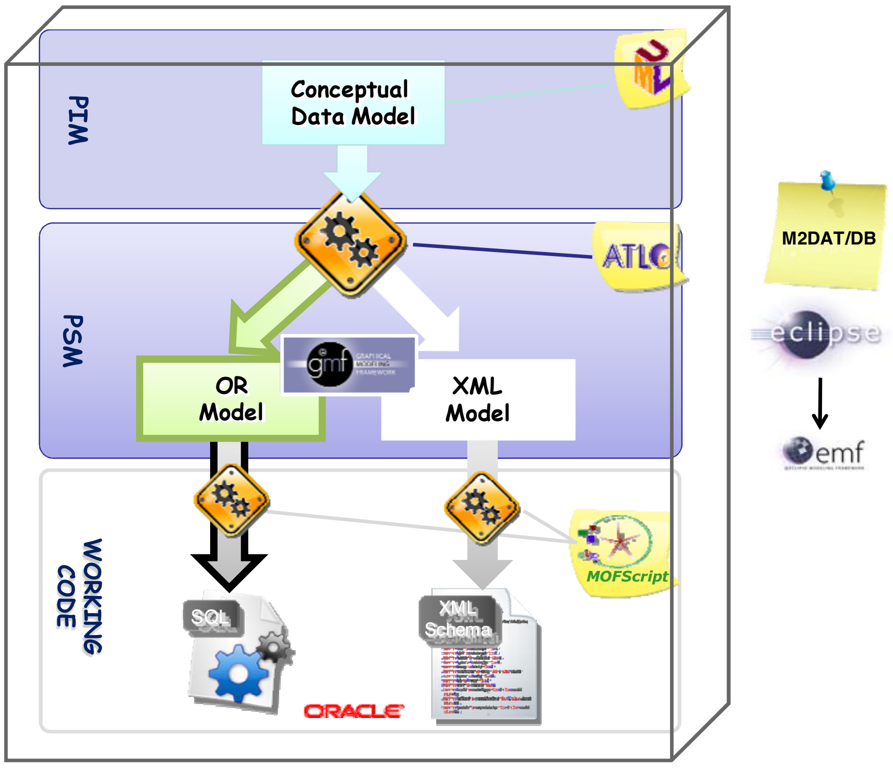


Fig. 1. MIDAS technical solution for ORDB Development: M2DAT/DB

that end. Finally, a MOFScript model to text transformation takes the previous OR DB model as input and generates the SQL code for Oracle that implements the modeled DB schema. We will present all these mappings in the following sections.

All this work is been integrated in M2DAT (MIDAS MDA Tool) as a new module: M2DAT/DB. M2DAT is a case tool whose aim is to support all the techniques proposed in MIDAS for the semiautomatic generation of the information system. The modular nature of the MIDAS architecture facilitates the modular development of M2DAT. Therefore, we are able to address the development of separate modules that implement the different proposals of MIDAS and later integrate them by means of a model transformation bus. The tool is now under development in our research group and its early functionalities have been already presented [24]. Fig. 1 shows the architecture of M2DAT/DB.

2.1 Modeling OR DB

In previous works we described a pair of UML profiles for OR DB modeling. They were defined according to the earlier specifications of the SQL standard and Oracle OR capabilities [12].

However, when we first addressed the task of implementing the PIM to PSM mappings of our proposal, we decided to shift from UML profiles to Domain Specific Languages (DSL) [14]. Technology is playing a key role in the distinction between UML based and non-UML based tools: the facilities provided in the context of the *Eclipse Modeling Project* (EMP) and other DSL frameworks, like the *Generic Modeling Environment* (GME) or the *DSL Tools*, have shifted the focus from UML-based approaches to MOF-based ones [5]. Therefore, regarding existing technology for (meta-) modeling and model transformations, it was more convenient to express the new concepts related with OR DB modeling by means of well defined metamodels. To that end we have defined a pair of MOF based metamodels. Here we will summarize only the one for Oracle10*g* since the one for the SQL standard is very similar.

Figure 2 shows the metamodel that represents the object characteristics of Oracle10*g*'s OR model. The main differences with regard to the standard metamodel are: Oracle neither supports the ROW type nor the inheritance of tables and Oracle does support the NESTED TABLE type instead of the MULTISET type (although they represent nearly the same concept).

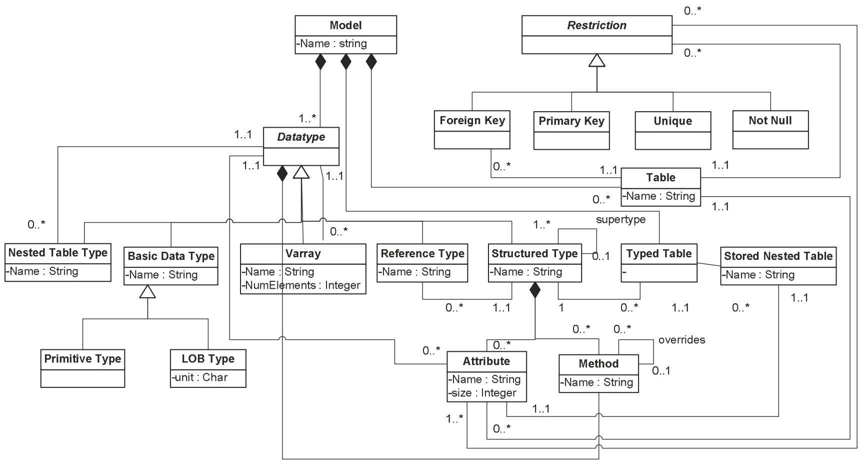


Fig. 2. Oracle 10g's OR metamodel

2.2 PIM to PSM Transformations for OR DB Development

We have to define a set of mapping rules to move from the conceptual data model (PIM) to the OR DB one (PSM). Regarding how they should be defined, in [19] it is stated that "the mapping description may be in natural language, an algorithm in an action language, or a model in a mapping language". This way, in previous works [7] we sketched a common approach to address the development of model to model transformations in MIDAS framework:

1. First, the mappings between models are defined using natural language.
2. Next, those mappings are structured by collecting them in a set of rules, expressed again in natural language.
3. Then, these mapping rules are formalized using graph grammars.
4. Finally, the resulting graph transformation rules are implemented using one of the existing model transformation approaches (we have chosen the ATL language; this decision will be explained later).

This proposal is oriented to give solution to some problems we have detected in the field of model transformations: there is a gap between the developers of the different model transformation engines and those who have to use them. We aim at reducing this gap by providing with a simple method for the definition of mappings. In this sense, the fact that models are well represented as graphs is particularly appealing to shorten the distance between modelers and model transformation developers. Moreover, formalizing the mappings before implementing them can be used to detect errors and inconsistencies in the early stages of development and can help to increase the quality of the built models as well as the subsequent code generated from them. These activities are especially important in proposals aligned with MDA (like the one of this paper), because MDA proposes the models to be used as a mechanism to carry out the whole software development process. Likewise the formalization of mappings has simplified significantly the development of tools supporting any model-driven approach.

We have followed this method to carry out the PIM to PSM mappings of our proposal for OR DB development. As a first step, we collected the mappings in the set of rules summed up in Table 1.

It is worth mentioning that these rules are the result of a continuous refining process. As mentioned before, the very first version of these mappings was conceived for the SQL:1999 standard and the 8*i* version of Oracle [12].

Table1. OR PIM to PSM Mappings

Data PIM		Standard Data PSM (SQL:2003)	Product Data PSM (Oracle10*g*)
Class		Structured Type + Typed Table	Object Type + Object Table
Class Extension		Typed Table	Table of Object Type
Attributes	Multivalued	Array/Multiset	Varray/Nested Table
	Composed	ROW/Structured Type (column)	Object Type (column)
	Calculated	Trigger/Method	Trigger/Method
Association	One-To-One	Ref/Ref	Ref/Ref
	One-To-Many	Ref/Multiset/Array	Ref/Nested Table/Varray
	Many-To-Many	Multiset/Multiset Array/Array	Nested Table/Nested Table Varray/Varray
	Aggregation	Multiset/Array	Nested Table/Varray of References
	Composition	Multiset/Array	Nested Table/Varray of Objects
	Generalization	Types/Typed Tables	Types/Typed Tables

The next step was the formalization of the mapping rules using a graph transformation approach [2]. Finally, those rules are implemented using the ATL Language [11], a model transformation language developed by ATLAS Group. It is mainly based on the OCL standard and it supports both the declarative and imperative approach, although the declarative one is the recommended.

We have chosen ATL because, nowadays, it is considered as a de-facto standard for model transformation since the OMG's Query/View/Transformations (QVT) practical usage is called into question due to its complexity and the lack of a complete implementation. There do exist partial implementations, both of QVT-Relational, like ikv++'s medini QVT, and QVT Operational Mappings, like SmartQVT or Eclipse's QVTo. However, none of them combines both approaches (declarative and imperative), in theory, one of the strengths of QVT, and they are still to be adopted by the MDD community. On the other hand, ATL provides with a wide set of tools and its engine is being constantly improved. Next section presents some of the mapping rules next to their implementation using the ATL language.

3 Case Study

This section presents, step by step, our proposal for model-driven development of OR DB by means of a case study: the development of an OR DB for the management of the information related to the projects of an architect's office.

It should be noticed that, once the PIM has been defined, the whole process is carried out in an automatic way. Nevertheless, the designer has the option to refine and/or modify the OR generated model before the code generation step. To ease this task, we have developed a graphical editor for models conforming to the OR metamodel sketched in section 2.1.

3.1 Conceptual Data Model

The first step in the proposed development process is to define the conceptual data model. Fig. 3 shows the model for our case study: a project manager is related to one or more projects. Both, his *cod_manager* or his name serve to identify him. He has an *address* plus several phone numbers. In its turn, every project has a name and it groups a set of plans, each one containing a set of figures. We would like to know how many figures a plan contains. Thus, a *number figures* derived attribute is included in the Figure class. Those figures could be polygons, which are composed of lines. Every line has an identifier and contains a set of points. The length of the line is obtained by computing the distance between their points.

We have used UML2 to define this model. UML2 is an EMF-based implementation of the UML metamodel for the Eclipse platform. We use UML2 for the graphical definition of PIM models in M2DAT.

3.2 PIM to PSM Mapping

Now we will show how some of the mapping rules are applied to obtain the OR PSM for Oracle 10*g*. We present the formalization of each rule with graph grammars next to its ATL implementation.

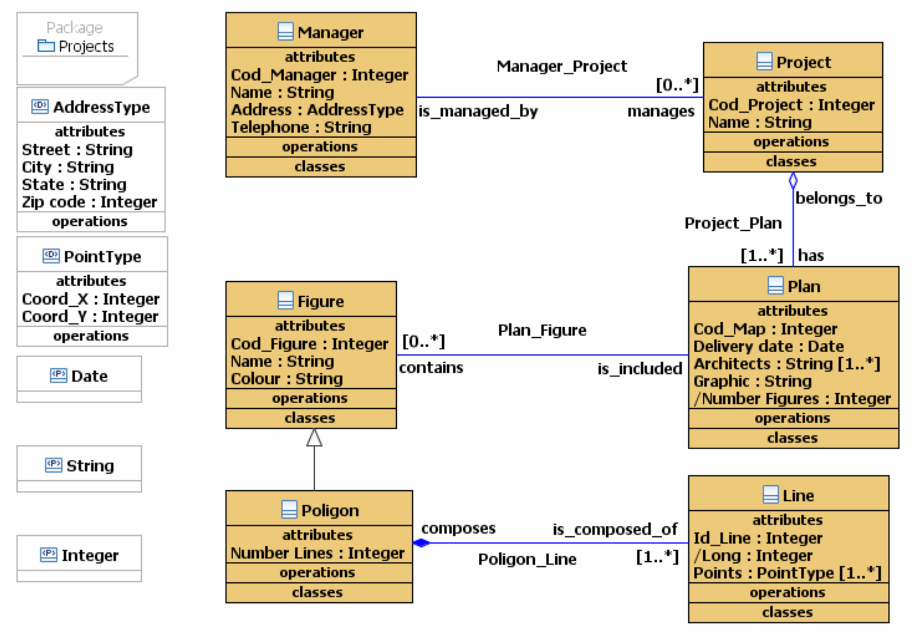
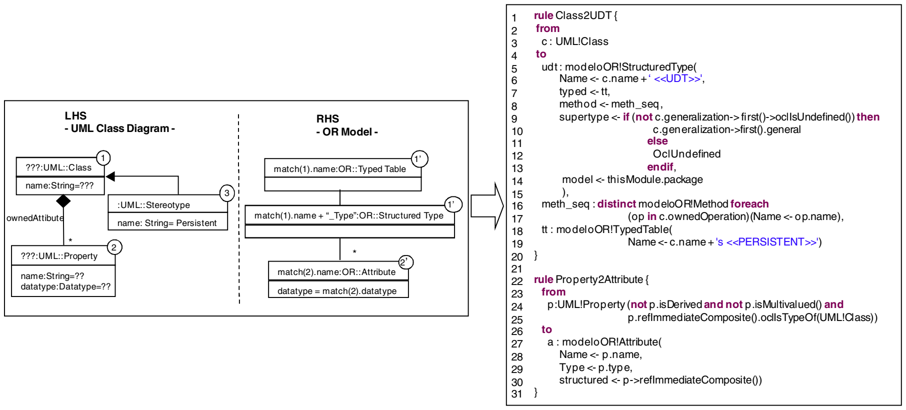


Fig. 3. Conceptual Data Model for the case study

Mapping Classes and Properties. Left-hand side of Fig. 4 shows the graph transformation rule to map persistent classes (PIM) to DB schema objects (PSM).



```
rule Class2UDT {
 from
   c : UML!Class
 to
   udt : modeloOR!StructuredType(
         Name <- c.name + ' <<UDT>>',
         typed <- tt,
         method <- meth_seq,
         supertype <- if (not c.generalization-> first()->oclIsUndefined()) then
                                c.generalization->first().general
                               else
                                OclUndefined
                               endif,
          model <- thisModule.package
          ),
   meth_seq : distinct modeloOR!Method foreach
                        (op in c.ownedOperation) (Name <- op.name),
   tt : modeloOR!TypedTable(
                        Name <- c.name + 's <<PERSISTENT>>')
}

rule Property2Attribute {
  from
    p:UML!Property (not p.isDerived and not p.isMultivalued() and
                        p.refImmediateComposite().oclIsTypeOf(UML!Class))
  to
    a : modeloOR!Attribute(
        Name <- p.name,
        Type <- p.type,
        structured <- p->refImmediateComposite())
}
```

Fig. 4. Persistent Classes mapping rule

Whenever an UML class stereotyped as persistent is found on the PIM (①–③), a structured type (also known as user-defined type - UDT) and a typed table are added on the PSM (①'). The type of the new table will be the UDT. Each property associated with the persistent class will be mapped by adding an attribute to the UDT (②⇨②').

Right-hand side of Fig. 4 shows the *Class2UDT* and *Property2Attribute* ATL rules that implement the prior graph transformation rule. The *Class2UDT* rule states that for every Class found in the source model (`UML!Class`), an UDT and a typed table are created in the target model (`modeloOR!StructuredType` and `modeloOR!TypedTable`). A set of direct bindings initialize the simple attributes of the structured type, such as the *name*. While more complex bindings serve to initialize the rest. For instance, the expression used to initialize the *supertype* attribute first checks if the source class was the descendant of any other class (lines 9-13).

On the other hand, the *Property2Attribute* ATL rule maps every UML property of each class in the source model to an attribute of the corresponding UDT. To that end, the *structured* property of each new attribute is bound to the source class that contains the property that is being mapped (line 30). When the ATL engine evaluates this expression, the reference to the source class is replaced by a reference to the structured type that maps the source class. This way, we do not need to navigate the target model when we need to establish references between its elements. It is the ATL engine that deals with this task using its resolve mechanism [11]: whenever a reference to an element in the source model is found, it is replaced by a reference to the element that maps it in the target model.

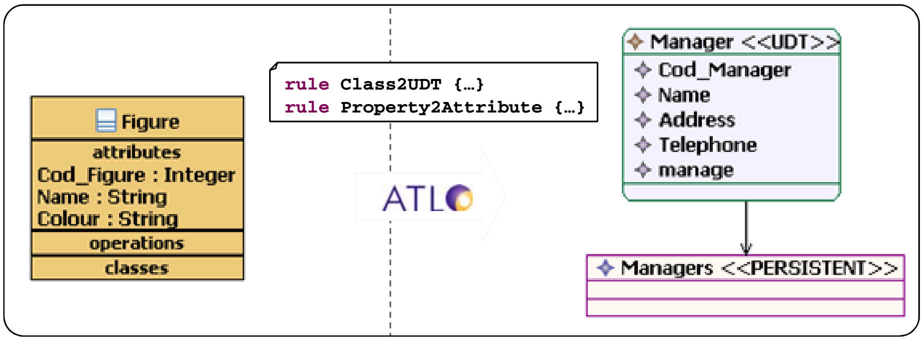
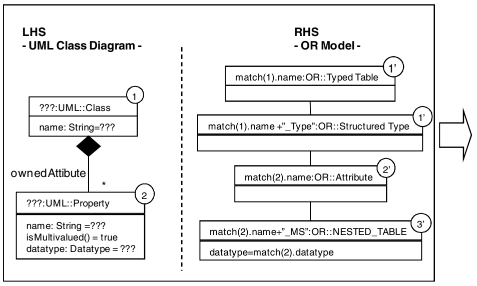


Fig. 5. Mapping of class *Manager*

Fig. 5 shows the instantiation of this rule. On the left-hand side there is an extract of the conceptual data model: the *Manager* UML class owning a set of properties. The right-hand side is a partial screen capture from our graphical editor. It shows the elements that map the source class into the target model: an object type named *Manager* owning a set of attributes, next to the corresponding *Managers* typed table.

Notice that we have opted for giving a UML-alike appearance to the graphical editor. This way, we try to take the best from UML: it is well-known by software engineers and developers. However, we try to take out the worst from UML: its complexity. When you are developing model transformations for UML stereotyped models you have to deal with the complex and big metamodel of UML – profiles always add something, they never remove anything. We prefer to avoid this complexity by using a DSL with "UML graphical syntax".

Mapping Multivalued Properties. The mapping rules we have just presented work fine for mapping classes and their properties in the generic case. However, as showed on Table 1, specific mapping rules are needed to handle the special nature of certain properties. This way, we have defined rules for mapping *multivalued*, *composed* and *calculated* (i.e. derived) properties. For the sake of space we present here just the first one (see Fig. 6).



```
1   rule PropertyMultivalued2Attribute {
2    from
3      p :UML!Property (not p.isDerived and p.isMultivalued() and
4                               p.refImmediateComposite().oclIsTypeOf(UML!Class))
5    to
6     a : modeloOR!Attribute(
7           Name <- p.name,
8           Type <- thisModule.generateNestedTableMultivalued(p),
9                               structured <-(p.refImmediateComposite())
10          ),
11    st : modeloOR!StoredNestedTable(
12         Name<- p.name + 's_ List',
13         attribute <- a,
14         typed <-thisModule.resolveTemp(
15                               p->refImmediateComposite(),'tt')
16       )
17  }
18  unique lazy rule generateNestedTableMultivalued {
19    from
20      p: UML!Property
21    to
22      nt: modeloOR!NestedTableType (
23         Name <-'NT_' + p.name + 's <<NT>>',
24         Type <- p.type,
25         model <- thisModule.package)
26  }
```

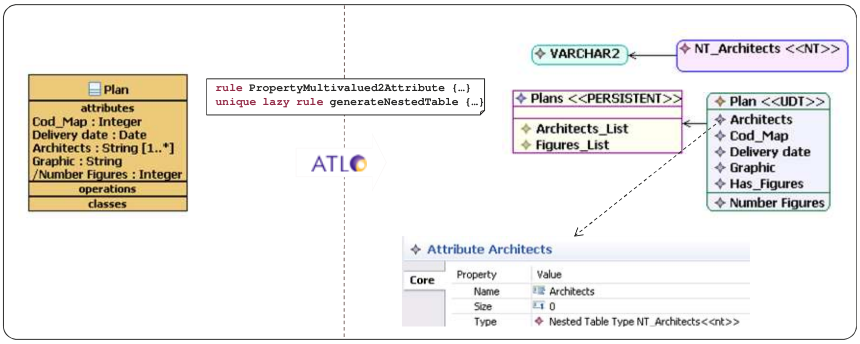
Fig. 6. Multivalued properties mapping rule

Multivalued properties on the conceptual data model correspond to columns of a collection type on OR DB schemas. Oracle 10*g* provides with two pre-defined constructors for collection types: VARRAY and NESTED TABLE. Fig. 6 shows the graph transformation rule when we choose the NESTED TABLE constructor.

The UML class (①) has a multivalued property (②). It is stated by the *true* value of its *isMultivalued* attribute. Therefore, the structured type that maps the persistent class (①') owns an attribute of a NESTED TABLE type (②'–②') (the same is valid for a VARRAY type).

When we coded this rule we did not know of any way to decide whether we want to create a VARRAY or a NESTED TABLE each time the rule is executed. Therefore, we chose NESTED TABLE as the default collection type since it is the less restrictive. Right now we have solved this problem using annotation models to parameterize the transformation. Right-hand side of Fig. 6 shows that we need two different rules to code the mapping of multivalued properties. The guard of the *Property2MultiValuedAttribute* rule ensures that only multivalued properties of UML classes will match this rule (line 3-4). Its target pattern specifies that two elements will be added in the target model to map the property found on the source model: an OR attribute and an OR Nested Table (lines 11-25). Moreover, by the time the new attribute is created, a Nested Table type is also defined. To that end, the *type* property of the new attribute is initialized by calling the *generateNestedTableMultivalued* lazy rule (line 8-9). A lazy rule is also a declarative rule but it has to be explicitly called. This way, the type of the attribute is the newly created Nested Table type.

Fig. 7 shows the result of mapping the multivalued property *Architects* of the class *Plan*.



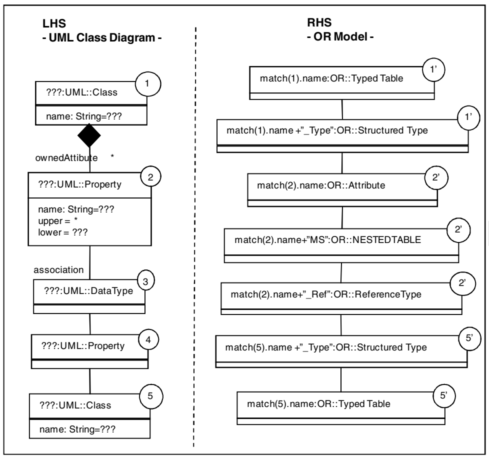
```
rule PropertyMultivalued2Attribute {…}
unique lazy rule generateNestedTable {…}
```

Property	Value
Name	Architects
Size	0
Type	Nested Table Type NT_Architects<<nt>>

Fig. 7. Mapping of *Architects* multivalued property

Mapping Associations. We propose different transformation rules depending on the maximum multiplicity of the classes involved in the association (see Table 1). For the sake of simplicity, we have considered only uni-directional relationships. The same rules can be applied for bi-directional relationships with two minor adjustments: duplicating the construction in both sides of the rule and adding some mechanism to maintain referential integrity (a trigger for instance). Here we introduce the rule for mapping many-to-many associations.

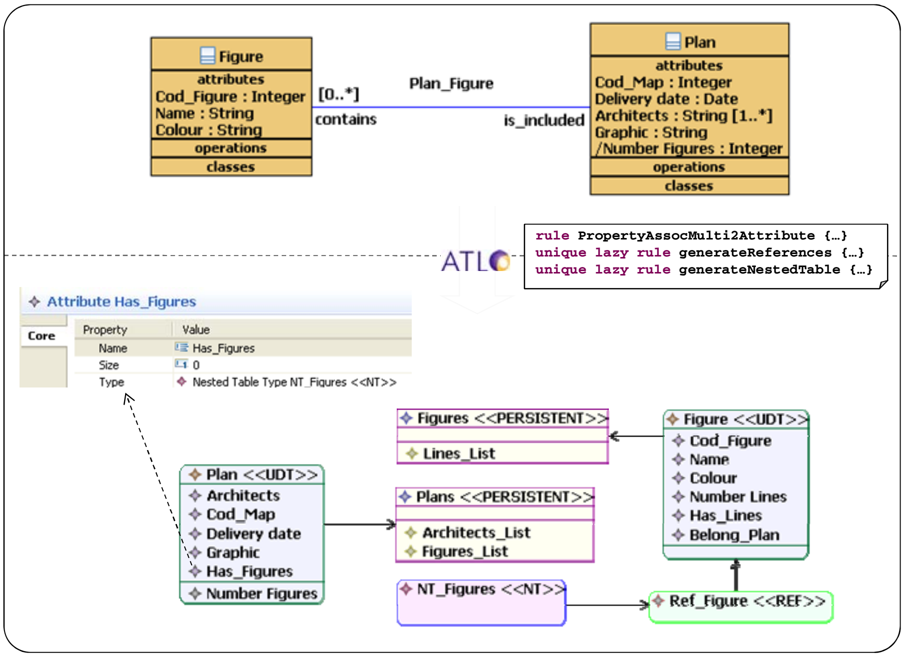
As left-hand side of Fig. 8 shows, *Maximum Many Multiplicity* associations are identified in the PIM by the value of the *upper* attribute in the UML property (②) of the source class. The corresponding uni-directional relationship in the PSM is built upon an attribute in the Structured Type that maps the source class of the association (①’–②’). This attribute is a collection of references to the Structured Type that maps the target class (②’–②’). Thus, it is a Nested Table object containing a set of references (REF type objects).


LHS
- UML Class Diagram -
???:UML::Class
name: String=???
ownedAttribute *
???:UML::Property
name: String=???
upper = *
lower = ???
association
???:UML::DataType
???:UML::Property
???:UML::Class
name: String=???
1
2
3
4
5
RHS
- OR Model -
match(1).name:OR::Typed Table
match(1).name +"_Type":OR::Structured Type
match(2).name:OR::Attribute
match(2).name+"MS":OR::NESTEDTABLE
match(2).name+"_Ref":OR::ReferenceType
match(5).name +"_Type":OR::Structured Type
match(5).name:OR::Typed Table
1'
1'
2'
2'
2'
5'
5'

```
rule PropertyAssocMulti2Attribute {
 from
  p:UML!Property(p.refImmediateComposite().oclIsTypeOf
                 (UML!Association) and p.isMultivalued() and
                          p.isChangeable())
 to
  a : modeloOR!Attribute(
     Name <- p.name,
     Type <- thisModule.generateNestedTable(p.type),
     structured <-p.getProperty().type),
  st: modeloOR!StoredNestedTable(
     Name <- p.name + '_List',
     attribute <- a,
     typed <- thisModule.resolveTemp
                 (p.getProperty().type,'tt'))
}
unique lazy rule generateNestedTable {
 from
   c:UML!Class
 to
   nt : modeloOR!NestedTableType(
      Name <- 'NT_' + c.name + 's <<nt>>',
      Type <- thisModule.generateReferences(c),
      model <- thisModule.package)
}
unique lazy rule generateReferences {
 from
   c:UML!Class
 to
   ref : modeloOR!ReferenceType(
      Name <- 'Ref_' + c.name + '<<REF>>',
      Type <- c,
      model <- thisModule.package)
}
```

Fig. 8. Associations mapping rule

Right-hand side of the picture shows the corresponding ATL implementation. The guard ensures that only UML multivalued properties of an UML association will be mapped by this rule (line 3-5). To map the property, an OR attribute is created (lines 6-16). The type of the attribute will be a Nested Table type. It is created by the *generateNestedTable* lazy rule (lines 17-25). Also, we need to create a new REF type by invoking the *generateReferences* lazy rule (lines 26-33). It serves to define the type of the elements of the Nested Table type. Notice that both lazy rules are *unique*. This way, we ensure that neither the REF, nor the Nested Table types will be duplicated if those rules are called again with the same parameters. Instead, they will return a reference to the already created types.



```
rule PropertyAssocMulti2Attribute {…}
unique lazy rule generateReferences {…}
unique lazy rule generateNestedTable {…}
```

Property	Value
Name	Has_Figures
Size	0
Type	Nested Table Type NT_Figures <<NT>>

Fig. 9. Mapping of *Has_Figures* association

Fig. 9 shows the result of mapping the association between a *Plan* and its *Figures*: apart from the structured types and the typed tables that map the *Plan* and the *Figure* classes, a REF type (*Ref_Figure*) and a Nested Table type are created (*NT_Figures*). The later contains a collection of objects of the former.

3.3 Code Generation

In order to implement the generation of the corresponding SQL code from the OR model, we have chosen the MOFScript language [18]. MOFScript is a prototype implementation based on concepts submitted to the OMG MOF Model to Text RFP process.

In front of the declarative approach of ATL (and the vast majority of existing model to model proposals), model to text transformation engines take the form of imperative programming languages. In fact, a MOFScript script is a parser for models conforming to a given metamodel. While it parses the model structure, it generates a text model based on transformation rules. On a second phase this text model is serialized into the desired code. This way, the script uses the metamodel to drive the navigation through the source model, just as an XML Schema drives the validation of an XML file. As a matter of fact, every model is persisted in XMI format, an XML syntax for representing UML-like (or MOF) models.

The program that implements the model to text transformation is basically a model parser. It navigates the structure of the OR model, generating a formatted output stream: the SQL script that implements the modeled DB schema in Oracle 10*g*. In the following we introduce this script showing some code excerpts. The reader is referred to [18] for more information on how to configure MOFScript execution.

As showed below, a *main* function (line 13) is the entry point for the script. It includes a set of rules for processing each possible type of element that can be found in the source model (so-called context types in MOFScript). Besides, we include the *eco* parameter in the script header to specify which the input metamodel is (line 11). To that end we use the URI that identifies the OR metamodel we have presented in section 2.1.

```
11 texttransformation codigo (in eco:"http:///modeloOR.ecore") {
12
13   eco.Model::main(){
```

```
18    //Structured Type code generation
19    self.datatype->forEach(s:eco.StructuredType)
20    {
21                s.generateStructured()
22    }
```

Next, a transformation rule is defined for each context type. For simple rules, we code the rule inside the *main* body whilst the complex ones are coded by means of auxiliary functions. Those functions are invoked from the *main* body.

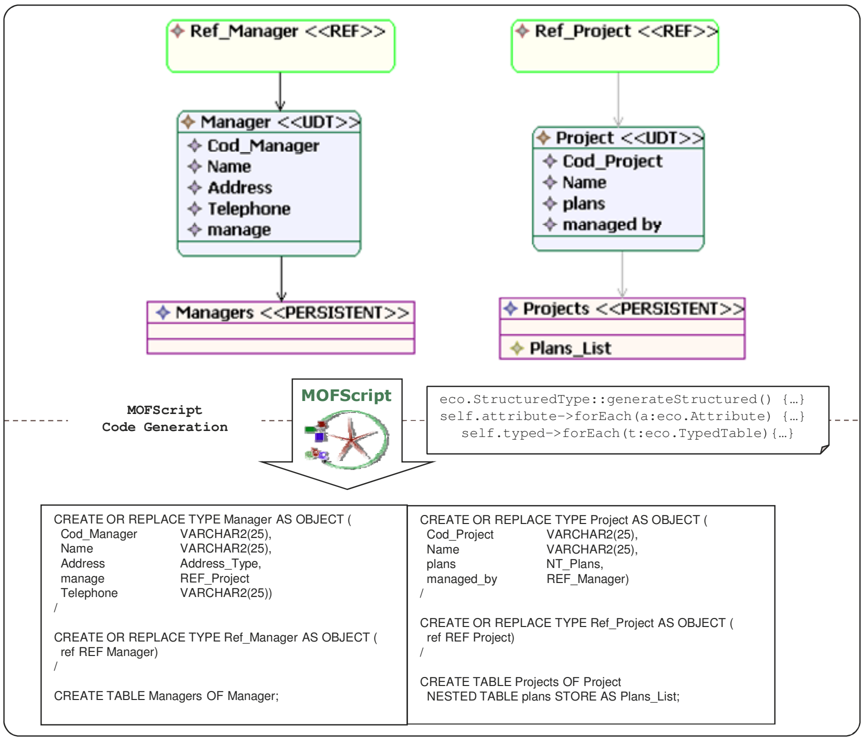
For instance, the rule for Structured Types creation is probably the most complex one since it encapsulates a lot of semantics. Thus, it is coded in the *generateStructured* auxiliary function. The *main* body invokes it for every Structure Type object found in the source model (line 21 below).

The next code extract presents the *generateStructured* function (for the sake of space, we have skipped the code for attributes mapping, lines 100-127).

```
88   // Auxiliary Function for Structured  Type code generation
89  eco.StructuredType::generateStructured () {
90     var texto:String=""
91     if (self.supertype.Name.size()=0)
92       texto="CREATE OR REPLACE TYPE" + self.Name.replace(" <<UDT>>", "") + " AS OBJECT \r\n("
93     else
94       texto="CREATE OR REPLACE TYPE" + self.Name.replace(" <<UDT>>", "") + " UNDER " +
95                 self.supertype.Name.replace(" <<UDT>>", "") + "\r\n("
96       println(texto)
97
98      // Attributes
99      self.attribute->forEach (a:eco.Attribute) {
...........................................................................................
128      // Typed Tables
129      self.typed->forEach (t:eco.TypedTable ){
130        println("CREATE TABLE " + t.Name.replace(" <<PERSISTENT>>","") +
                        " OF " + self.Name.replace(" <<UDT>>","") + "\r\n(")
131        tgenerateTypedTable ()
132        println("")
133      }
134   }
```

First, the auxiliary variable that will store the SQL code is initialized (line 90). Next, we add the SQL code to start the creation of the structured type, distinguishing those types that inherit from any other type (lines 91-96) from those that do not. Then, another rule is coded to transform attribute objects (lines 100-127). This rule is executed for every attribute object that belongs to the structured type (line 99). Finally, we iterate over the *typed* property of the structured type to identify the typed tables, whose type is the one being mapped (line 129). Then, the SQL code to start the creation of each table is added and the rule to create them, the *generateTypedTable* function, is invoked (lines 130-131).

To conclude this section, Fig. 10 shows a piece of the SQL code generated for the case study.



```
eco.StructuredType::generateStructured() {…}
self.attribute->forEach(a:eco.Attribute) {…}
  self.typed->forEach(t:eco.TypedTable){…}
```

```
CREATE OR REPLACE TYPE Manager AS OBJECT (
 Cod_Manager        VARCHAR2(25),
 Name               VARCHAR2(25),
 Address            Address_Type,
 manage             REF_Project
 Telephone          VARCHAR2(25))
/

CREATE OR REPLACE TYPE Ref_Manager AS OBJECT (
 ref REF Manager)
/

CREATE TABLE Managers OF Manager;
```

```
CREATE OR REPLACE TYPE Project AS OBJECT (
 Cod_Project        VARCHAR2(25),
 Name               VARCHAR2(25),
 plans              NT_Plans,
 managed_by         REF_Manager)
/

CREATE OR REPLACE TYPE Ref_Project AS OBJECT (
 ref REF Project)
/

CREATE TABLE Projects OF Project
 NESTED TABLE plans STORE AS Plans_List;
```

Fig. 10. Code Generation excerpt

The upper side is a screen capture of the developed graphical editor. It shows an extract from the OR model of the case study. Specifically the *Manager* and *Project* types and the corresponding typed tables, next to the REF types created from them. This code is generated by the execution of the mapping rules listed in the annotation beside.

4 Related Work

The mapping of object models to relational DB schemas and vice versa has been widely used as a case study to present new model transformations proposals. For instance, see the works from [6].

Besides there are some works on the application of model-driven techniques to Data Warehouses (DW) development though they are still too immature, limiting themselves to the proposal of new UML profiles for DW modeling. Probably the most mature are the ones from Mazón et al. [9, 15] and Tonkunaite et al. [23] where a modeling process is proposed and the corresponding model transformations are already defined. Nevertheless, they are just formally specified with the QVT standard but not implemented using any technical solution for model transformation.

Likewise, we should consider the works from Atzeni, Bernstein et al. on model management [1]. They propose a framework focused on schema mappings. The proposal is based on a solid formal basis (relational DBs) but some objections can be made from the point of view of MDE. First, the use of a new metametamodel (known-as Supermodel) different from MOF is not justified. It makes it hard to develop bridges towards the universe of MOF-compliant proposals. Moreover, such Supermodel is extended when needed, raising other questions. Where is the limit to extend the Supermodel? Why is it not extended for each new metamodel considered? Using an evolving metametamodel seems to be a bad practice. One can argue that, each time the Supermodel is extended, you might consider updating your existing metamodels to consider the modifications or just to improve them. But there is no doubt this is a bad practice. Evolving metamodels are a hard problem in MDE. Whenever you modify your metamodel, you will have to propagate those changes to your model transformations, code generator, diagrammers, etc. i.e. to all your tooling.

However, to the best of our knowledge there are no previous works applying model-driven techniques for OR DB development. As a matter of fact, this is the main contribution of this work with regard to other existing approaches for OR DB development, like the ones from [16] and [17]. On the one hand, the creation of the models that take into account technological aspects are postponed as much as possible in the development process, so that technology changes imply low costs. On the other hand, the definition of model to model and model to text transformations results in the automatic generation of the most part of the working-code. In this sense, our proposal is original and complete. We have defined a modeling process and we have developed the technical support to carry out the proposed process.

5 Conclusions and Future Work

In this paper we have used a case study to present the tool support of our proposal for model-driven development of OR DB schemas. To that end, we have implemented an ATL model transformation that generates an OR DB model from a conceptual data model and a MOFScript model to text transformation that generates the SQL code for the modeled DB schema. As part of the proposal we have defined a MOF-based DSL for OR DB modeling as well as a graphical editor for such DSL.

The implementation of our proposal is one of the modules of M2DAT (MIDAS MDA Tool). M2DAT is a case tool, which integrates all the techniques proposed in MIDAS for the semi-automatic generation of service-oriented Information Systems. We are still working on the development of the technical support for the rest of the methodological proposals comprised in MIDAS framework. In this sense, this work has served also as test bench. We have obtained a set of lessons learned and good practices to address such challenges. Once we complete the modules that rest, we will address the integration task.

Besides, this paper allows confirming one of the hypotheses around model transformation sketched in [7]: PIM to PSM transformations are easier to automate than PIM to PIM transformations. The former implies decreasing the abstraction level, thus handling more specific artifacts that result easier to model. The later implies a higher abstraction level and more similarity between the elements of the source and target models, in short, more ambiguities. Thus, it requires a higher level of decision making from the designer.

Regarding this issue there is still some place for improvement. At present time we are working in the use of weaving models [3] to annotate the OR model (i.e. the source model). This way, we are able to *parameterize* the model transformation, so that the mapping process for OR DB development can be customized. For each specific case some design decisions can be made. This way we keep the generic nature of the model transformation without polluting the source model.

Acknowledgments

This research has been carried out in the framework of the MODEL-CAOS (TIN2008-03582/TIN) and CONSOLIDER (CSD2007-0022, INGENIO 2010) projects financed by the Spanish Ministry of Education and Science (TIN2005-00010/) and the M-DOS project (URJC-CM-2007-CET-1607) cofinanced by the Rey Juan Carlos University and the Regional Government of Madrid.

References

1. Atzeni, P., Cappellari, P., Gianforme, G.: MIDST: model independent schema and data translation. In: Proceedings of the 2007 ACM SIGMOD international Conference on Management of Data. SIGMOD 2007, Beijing, China, June 11-14, pp. 1134–1136. ACM, New York (2007)
2. Baresi, L., Heckel, R.: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 402–429. Springer, Heidelberg (2002)
3. Bernstein, P.A.: Applying Model Management to Classical Meta Data Problems. In: First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA (2003)
4. Bertino, E., Marcos, E.: Object Oriented Database Systems. In: Díaz, O., Piattini, M. (eds.) Advanced Databases: Technology and Design. Artech House, Norwood (2000)
5. Bézivin, J.: Some Lessons Learnt in the Building of a Model Engineering Platform. In: 4th Workshop in Software Model Engineering (WISME), Montego Bay, Jamaica (2005)

6. Bézivin, J., et al.: Model Transformations in Practice Workshop. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 120–127. Springer, Heidelberg (2006)
7. Cáceres, P., De Castro, V., Vara, J.M., Marcos, E.: Model Transformations for Hypertext Modeling on Web Information Systems. In: SAC 2006. Proc. of the 2006 ACM Symposium on Applied Computing, pp. 1232–1239. ACM Press, New York (2006)
8. Chen, P.P.: The Entity-Relationship Model – Toward a Unified View of Data. ACM Transactions on Database Systems 1(1), 9–36 (1976)
9. Fernández-Medina, E., Trujillo, J., Villarroel, R., Piattini, M.: Developing secure data warehouses with a UML extension. Information Systems 32, 826–856 (2007)
10. ISO / IEC 9075-14:2008 Standard, Information Technology – Database Languages – SQL 2008, International Organization for Standardization (2008)
11. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
12. Marcos, E., Vela, B., Cavero, J.M.: Extending UML for Object-Relational Database Design. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 225–239. Springer, Heidelberg (2001)
13. Marcos, E., Vela, B., Cáceres, P., Cavero, J.M.: MIDAS/DB: a Methodological Framework for Web Database Design. In: Arisawa, H., Kambayashi, Y., Kumar, V., Mayr, H.C., Hunt, I. (eds.) ER Workshops 2001. LNCS, vol. 2465, pp. 227–238. Springer, Heidelberg (2002)
14. Marjan, M., Jan, H., Anthony, M.S.: When and how to develop domain-specific languages. ACM Comput. Surv. 37, 316–344 (2005)
15. Mazon, J.-N., Trujillo, J., Serrano, M., Piattini, M.: Applying MDA to the development of data warehouses. In: Proceedings of the 8th ACM international workshop on Datawarehousing and OLAP. ACM, Bremen (2005)
16. Muller, R.: Database Design for Smarties. Morgan Kaufmann, San Francisco (1999)
17. Naiburg, E.J., Maksimchuk, R.A.: UML for Database Design. Addison-Wesley, Reading (2001)
18. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., Berre, A.-J.: Toward Standardised Model to Text Transformations. In: Model-driven Architecture – Foundations and Applications, pp. 239–253 (2005)
19. Miller, J., Mukerji, J. (eds.): OMG. MDA Guide Version 1.0. Document number omg/2003-05-01 (retrieved 2003), http://www.omg.com/mda
20. Oracle Corporation. Oracle Database 10g. Release 2 (10.2), http://www.oracle.com
21. Selic, B.: The pragmatics of Model-Driven development. IEEE Software 20(5), 19–25 (2003)
22. Stonebraker, M., Brown, P.: Object-Relational DBMSs. In: Tracking the Next Great Wave. Morgan Kauffman, San Francisco (1999)
23. Tonkunaite, J., Nemuraite, L., Paradauskas, B.: Model driven development of data warehouses. In: 7th International Baltic Conference on Databases and Information Systems, July 3-6, pp. 106–113 (2006)
24. Vara, J.M., De Castro, V., Marcos, E.: WSDL automatic generation from UML models in a MDA framework. International Journal of Web Services Practices 1(1,2), 1–12 (2005)
25. Voelter, M.: MD* Best Practices (12/2008), http://voelter.de(retrieved January 10, 2009)

Typing in Model Management

Andrés Vignaga[1], Frédéric Jouault[2], María Cecilia Bastarrica[1], and Hugo Brunelière[2]

[1] MaTE, Department of Computer Science, Universidad de Chile
{avignaga,cecilia}@dcc.uchile.cl
[2] AtlanMod, INRIA Rennes Center - Bretagne Atlantique, Ecole des Mines de Nantes
{frederic.jouault,hugo.bruneliere}@inria.fr

Abstract. Model management is essential for coping with the complexity introduced by the increasing number and varied nature of artifacts involved in MDE-based projects. Global Model Management (GMM) addresses this issue enabling the representation of artifacts, particularly transformation composition and execution, by a model called a megamodel. Typing information about artifacts can be used for preventing type errors during execution. In this work, we present a type system for GMM that improves its current typing approach and enables formal reasoning about the type of artifacts within a megamodel. This type system is able to capture non-trivial situations such as the use of higher order transformations.

1 Introduction

In the field of software development, the increasing use of Model-Driven Engineering (MDE) in the past years has lead to more and more complex situations. Indeed, MDE mainly suggests basing the software development and maintenance process on chains of model transformations. A single transformation is often quite easy to handle but, as soon as industrial use cases are tackled, we are faced with large sets of MDE artifacts (e.g., models, metamodels, transformations) from which a solution have to be assembled. Thus, in order to be able to use them, but without unintentionally increasing the complexity of MDE, we need to invent new ways of creating, storing, viewing, accessing, modifying, and using the metadata associated with all these modeling entities. This is the purpose of Global Model Management (GMM) [6].

As the managed modeling resources may be of varied natures, some support for efficiently organizing them is required. In order to cope with this heterogeneity, a GMM solution has to rely on an architecture which allows precisely typing all the involved entities and corresponding relationships. This should prevent type errors during execution, such as the attempted execution of a non-transformation, or the use of a transformation on arguments for which it is not defined.

Currently, our GMM approach assumes that all managed artifacts are models conforming to precise metamodels. Model typing is then simply based on the conformance relationship, and metamodels are used as types. Moreover, artifacts are also related by strong semantic links. For instance, a transformation refers to its source and target metamodels (i.e., its parameter and return types). Information based on this typing approach suffices for most common cases. However, this scheme notably fails when transformations explicitly depend on these semantic links like in the two following cases: *a*) when a metamodel (i.e., a type used as a value) is used as input to a transformation, and *b*) when a transformation is used as input to another transformation (i.e., a function used as a value). Under these circumstances, it may not be possible to automatically infer a complete type for some elements. For this reason, a more complex typing approach is required.

In this paper we present a first version of a static type system dedicated to GMM. Understanding transformations as functions on models, we introduce a predicative formal system with dependent types with infinite hierarchies of sorts. The type system builds on the existing solution and features dependent products. These types are powerful enough for overcoming the identified limitations. Expressing GMM elements as terms of our calculus enables to statically type check these elements in a mechanical fashion.

This paper is organized as follows. Section 2 describes the GMM approach to model management, characterizes the limitations of the current version, and introduces a simple example illustrating them. Section 3 details our formal system by providing the syntax of terms and types, type judgments, as well as the set of type rules that form the type system. Section 4 revisits the example in order to demonstrate the application of the type system, and discusses its prototypical implementation within a tool that realizes GMM. Section 5 discusses related work. Section 6 concludes.

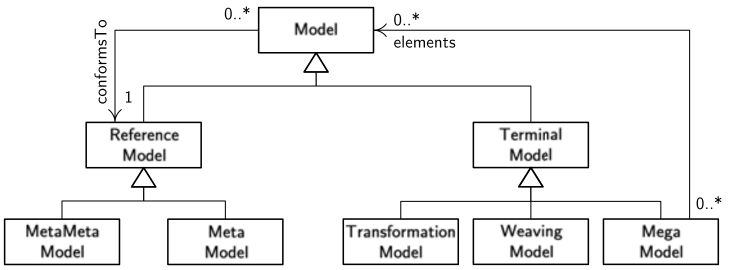
2 Global Model Management

In this section we summarize the basic concepts of GMM that enable an understanding of the general context, we discuss how typing is currently addressed and its limitations. For illustrating these issues we also discuss a small example, which will be revisited later on after our solution is presented.

2.1 Global Model Management Conceptual Framework

A Global Model Management approach is based on several general concepts (see Fig. [1]), which can be mapped to any concrete case. Most of these concepts, corresponding to a generic conceptual MDE framework, have already been presented in [5]. In addition to these, the concept of a *megamodel* is introduced here as a building block for *modeling in the large* [6]. The principle is the following: for each real-world complex system or process, there can be a *megamodel* [3] representing the different artifacts involved (i.e., models) and their relationships by specifying associated metadata. The type of an artifact or relationship between some artifacts, the identifier of a given artifact and its location, etc., are examples of such registered metadata.

MDE approaches generally introduce the following three different kinds of models, which occur in the conceptual framework of GMM illustrated in Fig. [1]:



```
context MetaMetaModel inv: self.conformsTo = self
context MetaModel inv: self.conformsTo.oclIsKindOf(MetaMetaModel)
context TerminalModel inv: self.conformsTo.oclIsKindOf(MetaModel)
```

Fig. 1. Global Model Management Conceptual Framework

– *terminal models* (M1) conform to *metamodels* and are representations of real-world *systems*.
– *metamodels* (M2) conform to *metametamodels* and define domain-specific concepts.
– *metametamodels* (M3) conform to themselves and provide generic concepts for metamodel specification.

Several kinds of *terminal models* may be considered, for example, *weaving models*, and *transformation models*. A *megamodel* is also a specific kind of terminal model, whose elements represent models themselves, as well as relationships between them. As it is a terminal model, a megamodel conforms to a specific metamodel: the *metamodel of megamodel*. If represented as models, available tools, services and service parameters may also be managed by a megamodel. There are actually many events that may change a megamodel like the creation or deletion of a model or metamodel, or the execution of a given transformation.

In addition, the current GMM framework proposes different kinds of relationships between them. The *model transformation* relationship allows specifying source and target metamodels of a given model transformation, and can thus be regarded as its signature. From an execution point of view, the *transformation record* relationship offers a way of representing the metadata needed for any potential execution of a given transformation. This allows specifying its actual input and output models.

To summarize, a megamodel can be viewed as a metadata repository where precise representations of models and links between them are stored and made available to users for various and varied purposes. In particular, the framework should be able to represent type information for adequately typing each element in a megamodel, and provide precise directions on how to use that information.

2.2 Limitations of the Current Typing Approach

As mentioned in the introduction, the current typing solution in GMM follows a simple approach: in principle, all entities are models. Each model conforms to a concrete metamodel, which is its type. The has-type relation (denoted by a colon ':') is therefore defined as follows: $conformsTo(m,M)$ iff $m{:}M$, for any model m and metamodel M. However, GMM involves other elements different from entities: relationships. Some elements have dual representations, for example a transformation may be regarded as a relationship (i.e., model transformation) but also as a model (i.e., transformation model) [4]. In the latter case, the type of the element is the metamodel it conforms to. For ATL (AtlanMod Transformation Language) transformations [11,12], this type is plainly *ATL*, which does not carry information about source and target types. In the former case, the relationship is actually unidirectional and thus a model transformation is understood as a function on models. Type information associated to the element adequately refers to the type of source and target models. However, such models are typed as models whether they are transformations or not, which may cause a loss of type information.

In GMM, typing information plays an important role. The results of a transformation must be properly typed for a later use. This is especially critical when the result is another transformation. For transformations which do not handle dual elements the current typing approach is adequate.

As transformations operate on models, it is natural to type their inputs and outputs as models. The problem is that some of them may have a dual representation and thus the typing approach we initially described does not suffice. The most common case is that of a higher-order transformation (HOT). Consider a transformation h that produces another transformation t. Here h is considered as a transformation, but t is considered as a model. As a consequence of this situation they are typed differently. The type of h refers to the types of its source and target elements. In particular, the type of the target element is the type of t, which is the metamodel t conforms to (e.g., *ATL*). The type of t as a relationship does not fit into this scheme and thus t is only partially typed. We do know that it is a transformation, but we do not know the types of its source and target elements.

This typing approach presents an interesting benefit though. Some form of genericity is introduced: a HOT taking an ATL transformation as source accepts any model conforming to *ATL* (i.e., *any* ATL transformation), regardless of the number and type of its source and target elements. This capability is something we would like to preserve.

Another situation where typing by metamodel may lead to a loss of type information is when transformations operate on metamodels. In fact, the type of any metamodel received or generated by one of such transformations is a metametamodel (e.g., *KM3* [5]). Then, from the type of the transformation, it is possible to know that a metamodel is involved, but not which one. If this metamodel is used for typing other models involved in the same transformation, then it may not be possible to type that transformation.

When the two situations described so far meet, even harder problems arise. The *KM32ATLCopier* transformation [2] is one simple yet interesting case for illustrating these issues. This HOT receives a metamodel M and produces an identity transformation (called a copier), which is specifically applicable to models conforming to *M*. The type of the resulting copier transformation clearly depends on *M*. Type information about *KM32ATLCopier* as a relationship may be found in the header of its ATL definition:

```
create OUT : ATL from IN : KM3
```

This type information for *KM32ATLCopier* is insufficient. First, *M* is not associated to the type of model `IN`. Second, all we know about model `OUT` from its type is that it is a transformation. Third, it is not possible to specify that we know that both its source and target models conform to *M*.

By introducing other kinds of types such as function types and parametric types, we will be able to deal with these issues. In Sect. 4 this example will be revisited and a type for *KM32ATLCopier* carrying richer information will be discussed.

3 A Type System for GMM

Our solution is based on a typed calculus called cGMM. We define a mapping between GMM constructs and terms and types of that calculus, and then we define a type system for it. Expressing elements within a megamodel as terms enables statically typechecking those elements in a mechanical fashion. This is the main concern of our work, which does not aim to be considered as a complete formalization of GMM.

The cGMM calculus is a predicative typed λ-calculus with dependent types similar to the underlying language of Coq [17], the Predicative Calculus of (Co)Inductive Constructions (pCIC) [14,19]. Although other approaches may be applicable, a functional one appropriately fits our needs. Like in the current implementation of GMM, we use typed variables for representing models conforming to a metamodel. Dependent products enables (dependent) function types for typing transformations, and universally quantified types for coping with genericity. In particular, higher-order functions naturally represent higher-order transformations. In addition, an infinite hierarchy of universes supports the notion of Type being a type (i.e., Type:Type), and enables a proper representation of the three kinds of models (M1 to M3) discussed in Sect. 2.1.

In order to formalize the type system we need to present some elements of our calculus first. We start by discussing its syntax and the mapping to GMM constructs, and then we address its typing.

3.1 Textual Syntax

All objects handled in cGMM have a type. Unlike most type theories, we do not make a syntactic distinction between types and terms because the type-theory itself forces terms and types to be defined in a mutually recursive way. We therefore define both types and terms in the same syntactical structure.

Sorts. Types are seen as terms and as such they should be typed. The type of a type is called a *sort*. In principle, we use types for typing models so we introduce the sort Type which intends to be the type of such types. Since sorts can be manipulated as terms they also should be given a type. Typing Type with itself leads to undecidable type systems [7]. As a consequence we need to introduce infinite sorts by means of a hierarchy of universes Type_i for any natural i. Thus our set of sorts $\mathcal{S}$ is defined by: $\mathcal{S} \equiv \{\mathsf{Type}_i | i \in \mathbb{N}\}$. These sorts satisfy the following property: $\mathsf{Type}_i{:}\mathsf{Type}_{i+1}$. In this way we understand Type_0 as the type of all metamodels (e.g., *Class* : Type_0), which turns Type_0 into a metametamodel. As in Coq, when referring to the universe Type_i the user will never mention the index i, which is managed by the system. Therefore from a user perspective Type:Type is safely assumed. Consequently, without indexes, Type is a metametamodel which conforms to itself, as required by the first constraint in Fig. 1. However, GMM is expected to support multiple metametamodels at the same time, for example KM3, Ecore, and so on. To that end, we define a separate hierarchy of universes for each of metametamodel, and a corresponding hierarchy should be included in $\mathcal{S}$ each time a new metametamodel is incorporated. In what follows we do not replicate our presentation for every possible metametamodel, rather, we refer to Type as an arbitrary metametamodel.

Terms. Terms are built from variables, several forms of dependent products, several forms of abstractions, applications, cartesian products, tuples, and projections. Assuming x is a variable and T, U are terms, cGMM terms are as follows:

Type	A sort, the type of all types
x	A variable
$\lambda x{:}T;U$	An abstraction (for type abstractions)
$\lambda x{:}T.U$	An abstraction (for classical λ-abstractions)
$\lambda x_1{:}T.x_2{:}U$	An abstraction (for functions with a constant result)
$\forall x{:}T.U$	A dependent product (for universal quantification)
$x{:}T \rightarrow U$	A dependent product (for dependent function types)
$T \rightarrow U$	A non dependent product (for function types)
$(T\ U)$	An application (for both functional application and type instantiation)
$U_1 \times \ldots \times U_n$	A cartesian product
$\langle T_1, \ldots, T_n \rangle$	A tuple
$\mathrm{T}\vert_i$	A projection

Type is a metametamodel and as such belongs to M3. Variables map to models, either at M1, M2 or M3. If a variable is typed by Type it represents a reference model, which is an element of M2 or M3. If it is typed by a term typed by Type, then it denotes a terminal model, which is an element of M1. A type abstraction is used for parameterizing types. Transformation models are represented as functions. GMM manages two kinds of transformations. On the one hand, transformation models can be externally defined in a suitable transformation language, such as ATL, and are thus seen as opaque operations on models where their internal definition is not accessible by the GMM environment. We call this kind of transformations *atomic* transformations. Currently, the only external transformation language supported by GMM is ATL through the GMM4ATL extension. In this work we assume that all atomic transformations are defined in ATL. On the other hand, transformations can be defined within a megamodel, using the language provided by the GMM4CT extension, as external compositions of other existing transformations, regardless of their kind. We call them *composite* transformations and they are model transformations (i.e., relationships) and not transformation models (i.e., entities). For representing first-order atomic transformations we use a special form of abstraction which returns a constant value. This variant was introduced for simplicity, because the only interesting information about the result of such transformations is its type. Therefore, the body of the corresponding function should be nothing but an arbitrary value of the right type. Since each function would require the inclusion of a suitable variable in the typing environment, which should be then accessed for retrieving its type, the environment would be bloated with this kind of definitions. With this abstraction, the variable representing the result is locally defined in the abstraction and therefore the term stays closed and no access to the environment is required. Other forms of transformations are represented by ordinary abstractions. For example, defining two atomic transformations like $\mathit{Class2Relational} \equiv \lambda y{:}\mathit{Class}.r{:}\mathit{Relational}$ and $\mathit{Relational2SQL} \equiv \lambda z{:}\mathit{Relational}.s{:}\mathit{SQL}$, it is possible to define a composite transformation $\mathit{Class2SQL} \equiv \lambda x{:}\mathit{Class}.(\mathit{Relational2SQL}\ (\mathit{Class2Relational}\ x))$.

A universal quantification represents the parametrization of a term with respect to a typed variable. This is usually used in conjunction with higher-order transformations for achieving genericity. Function types, which are just either dependent or non dependent functional views of products, are used for typing transformations. The non dependent case is the typical function type. In turn, in the dependent case, the target type depends on a value of the source type. A detailed example of this involving a HOT will be presented in the next section. An application on a parametric term allows its instantiation to a given type. An application on a function then maps to a transformation record and represents the execution of a transformation on a specific model.

Finally, a cartesian product enables a function type with multiple sources and multiple targets. A tuple is a sequence of typed terms, and projections extract a component from a tuple. As a remark, a HOT is a transformation that operates and/or produces other transformations. Thus a HOT is expressed as a function which either: *a*) has a parameter typed by a function type, or *b*) its body is a function. Free variables and substitution are defined as usual for λ-calculi. Substituting a term T to free occurrences of a variable x in a term U is denoted as $U\{x/T\}$.

3.2 Typing

A type system is a collection of type rules, however they are always formulated with respect to a static typing environment for the program fragment being checked. A *static typing environment* records the type of free variables during the processing of program fragments. For example, the has-type relation $a{:}A$ is associated with a static typing environment Γ that contains information about free variables of a and A.

Judgments. The description of a type system starts with the description of a collection of judgments of the form $\Gamma \vdash \mathcal{A}$ where Γ is a static typing environment, $\mathcal{A}$ is an assertion, and the free variables of $\mathcal{A}$ are declared in Γ. The static typing environment can be understood as a list of distinct typed variables such as $\varnothing, x_1{:}A_1, \ldots, x_n{:}A_n$. A static typing environment then maps to the notion of megamodel. The empty environment is denoted by $\varnothing$. The form of $\mathcal{A}$ determines the different judgments to be used within a type system. For our system, we need the following judgments:

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash T{:}U$	T is a well-formed term of type U in Γ

A judgment can be regarded as *valid* or *invalid*. Validity formalizes the notion of well typed programs and is based on type rules. Type rules are used to carry out step-by-step deductions, i.e., type derivations, which formally prove that judgments are valid.

Type Rules. Type rules may be organized according to their conclusion judgment. We distinguish environment well-formedness rules, whose names are of the form (Env ...), from term type rules. In turn, the latter group may be further organized in rules where terms are given the type Type (Type ...) and all the rest. Figure 2 shows some selected rules, where some of which are used further on the next section in our example.

The rule (Env $\varnothing$) is an axiom stating that an empty environment is a valid environment. This means that an empty megamodel is a valid megamodel. The rule (Env Var) extends an environment with a new variable provided that the variable is not defined in the environment and its type is a valid type. This corresponds to adding a new model to a megamodel. The rule (Type Ax) formalizes the property which holds for universes in the same hieararchy within $\mathcal{S}$. The rule (Var) extracts an assumption from an environment, that is, this allows us to use a model included in a valid megamodel.

Rules (Type DFun) and (Type Fun), construct dependent and non dependent function types respectively. In turn rules (Abs Par), (Abs DFun) and (Type Fun) type abstractions. These are type parametrization, dependent and non dependent functions respectively. Finally, rules (App TIns), (App DFun) and (App Fun) introduce applications. In the first case, it is a type instantiation, the others correspond to functional applications.

In the next section we revisit the example of Sect. 2 in detail and present a type derivation which applies many of the rules discussed above.

Soundness. The purpose of a type system is to prevent programs from causing type errors during their execution. A type system is *sound* when only well typed

(Env ∅)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env Var)
$$\frac{\Gamma \vdash T{:}s \quad s \in \mathcal{S} \quad x \notin \Gamma}{\Gamma, x{:}T \vdash \diamond}$$

(Type Ax)
$$\frac{\Gamma \vdash \diamond \quad i < j}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_j}$$

(Var)
$$\frac{\Gamma', x{:}T, \Gamma'' \vdash \diamond}{\Gamma', x{:}T, \Gamma'' \vdash x : T}$$

(Type DFun)
$$\frac{\Gamma \vdash T{:}\mathsf{Type}_i \quad i \leq k \quad \Gamma, x{:}T \vdash U : \mathsf{Type}_j \quad j \leq k}{\Gamma \vdash x{:}T \to U : \mathsf{Type}_k}$$

(Type Fun)
$$\frac{\Gamma \vdash T{:}\mathsf{Type}_i \quad i \leq k \quad \Gamma \vdash U : \mathsf{Type}_j \quad j \leq k}{\Gamma \vdash T \to U : \mathsf{Type}_k}$$

(Abs Par)
$$\frac{\Gamma \vdash \forall x{:}T.U : s \quad s \in \mathcal{S} \quad \Gamma, x{:}T \vdash t{:}U}{\Gamma \vdash \lambda x{:}T;t : \forall x{:}T.U}$$

(Abs DFun)
$$\frac{\Gamma \vdash x{:}T \to U : s \quad s \in \mathcal{S} \quad \Gamma, x{:}T \vdash t{:}U}{\Gamma \vdash \lambda x{:}T.t : x{:}T \to U}$$

(Abs Fun)
$$\frac{\Gamma \vdash x{:}T \to U : s \quad s \in \mathcal{S}}{\Gamma \vdash \lambda x{:}T.t{:}U : T \to U}$$

(App TIns)
$$\frac{\Gamma \vdash t : \forall x{:}U.T \quad \Gamma \vdash u{:}U}{\Gamma \vdash (t\ u) : T\{x/u\}}$$

(App DFun)
$$\frac{\Gamma \vdash t : x{:}U \to T \quad \Gamma \vdash u{:}U}{\Gamma \vdash (t\ u) : T\{x/u\}}$$

(App Fun)
$$\frac{\Gamma \vdash t : U \to T \quad \Gamma \vdash u{:}U}{\Gamma \vdash (t\ u) : T}$$

Fig. 2. Sample type rules of cGMM

programs execute without type errors [8]. This property of a type system is demonstrated by means of a soundness theorem. A proof of soundness rests upon the semantics of the underlying language, and other properties such as subject reduction and strong normalization. A full proof would deserve a paper on its own, as in [21]. Instead, we rely on the fact that cGMM is based on pCIC, which enjoys such properties [17]. In what follows we discuss some concepts related to those properties which lead to additional rule for our calculus.

Subject reduction, or type preservation, states that reductions preserve type and is formulated as follows. If $\varnothing \vdash T : U$ and $T \rhd T'$ then $\varnothing \vdash T' : U$ (which means that if T reduces then it does so to a value of type U). Note that this is not sufficient for type soundness because it does not rule out the case in which T has a type but it does not reduce. Additionally, the type system should not be able to type terms that cause type errors. In systems such as pCIC all reductions for all typable terms do terminate. Such terms are called strongly normalizing.

The fundamental rule that defines reduction $\rhd$ identifies the application of a function to a given argument with its result. This is called β-reduction and the rule is: $\Gamma \vdash ((\lambda x{:}T.t)\ u) \rhd t\{x/u\}$. In systems with dependent types, type conversion is additionally required. Type convertibility $T = U$ is achieved when terms T and U reduce to the same normal form. This enables a rule which says that two convertible well-formed types have the same inhabitants. In this way, terms of a type before a reduction are also typed by the type resulting from the reduction.

As a specific characteristic of cGMM, it is worth noting that an application of the form $((\lambda x_1{:}T.x_2{:}U)\ u)$ trivially reduces to $x_2{:}U$ in one step since no substitution is actually performed. In GMM, all transformations, even composite ones, are ultimately built in terms of atomic transformations like the one just discussed. This suffices for seeing that in cGMM well typed applications reduce to a value.

4 Application

In this section we demonstrate the application of the type system revisiting the *KM32ATLCopier* example, and discuss the prototypical implementation of the type system and its integration into the AM3 tool.

4.1 Example Revisited

Each element in a megamodel represented by an environment Γ is expressed as a term t of our calculus. Then, for a proper type T it should be possible to derive, using the type rules, a proof for $\Gamma \vdash t : T$. If this is possible, term t is well typed and T is its type. For the *KM32ATLCopier* transformation example we use $\mathsf{KM3}$ as a concrete metametamodel instead of Type, and define a term as follows:

$$\mathit{KM32ATLCopier} \equiv \lambda M{:}\mathsf{KM3}.\lambda x{:}M.y{:}M$$

This definition can be interpreted as "a transformation for which, given a metamodel M we get a transformation that takes a model conforming to M as source produces a model conforming to M as a target". At the outer level the term is an abstraction on variable M. The term at the inner level (i.e., the result of the outer term) is a function with a constant result which represents an atomic copier; its argument x is of type M and the result y is of type M as well. The purpose of the term is just to enable a proper typing for *KM32ATLCopier* and not to model its definition. If that was the case, then the copier would have been written as $\lambda x{:}M.x$, for emphasizing that the result is actually argument x, but in our context it is not necessary even though it would have been possible in this particular case. Now we can prove the following judgment which types *KM32ATLCopier*:

$$\varnothing \vdash \mathit{KM32ATLCopier} : M{:}\mathsf{KM3}{\rightarrow}M{\rightarrow}M$$

This type is a dependent function type on value M. Its co-domain is a non dependent function type where both the domain and the co-domain are M. The static typing environment is the empty environment, which means that no other elements are required within the megamodel for this definition to be meaningful. Next we show a type derivation that proves the well typing of the *KM32ATLCopier* definition.

We start by applying rule (Abs DFun) and as a result we have three subgoals: (1) $\varnothing \vdash M{:}\mathsf{KM3}{\to}M{\to}M : \mathsf{KM3}$, (2) $\mathsf{KM3} \in \mathcal{S}$, which naturally holds, and (3) $\varnothing, M{:}\mathsf{KM3} \vdash \lambda x{:}M.y{:}M : M{\to}M$. Proceeding with subgoal (1), we apply rule (Type DFun) for getting the following subgoals: (4) $\varnothing \vdash \mathsf{KM3}{:}\mathsf{KM3}$ which we can prove by applying rule (Type Ax) and rule (Env $\varnothing$), and then subgoal (5) $\varnothing, M{:}\mathsf{KM3} \vdash M{\to}M : \mathsf{KM3}$. We prove (5) by applying (Type Fun), which introduces the following subgoal twice: (6) $\varnothing, M{:}\mathsf{KM3} \vdash M{:}\mathsf{KM3}$. This is proved by succesively applying rules (Var), (Env Var), and (Type Ax). For proving (3) we apply rule (Abs Fun) which introduces subgoals: (5) and (2) again. This concludes the proof.

Type inference concerns algorithms that find types (if existing) for typing type-annotated terms, such as the term *KM32ATLCopier* defined before. However, if terms are constructed in a bottom-up fashion, that is, following a derivation from the leaves to the root, and the type information is properly preserved at each step, then the resulting term will be well typed by construction. As discussed next, this is the approach we follow in the implementation of the type system.

For concluding the example, we instantiate the *KM32ATLCopier* term for obtaining a specific copier transformation. To that end, we define the following term based on the *SQL* metamodel:

$$SQLCopier \equiv (KM32ATLCopier\ SQL)$$

This term is simply a functional application. The result should be a transformation from *SQL* to *SQL* as we prove next for the following judgment:

$$\varnothing, SQL{:}\mathsf{KM3} \vdash SQLCopier : SQL{\to}SQL$$

Since substitution $M{\to}M\{M/SQL\}$ produces $SQL{\to}SQL$, the proof simply consist of applying rule (App DFun) which produces a subgoal which is exactly the same judgment we previously proved, and subgoal $\varnothing, SQL{:}\mathsf{KM3} \vdash SQL{:}\mathsf{KM3}$, which is proved as subgoal (6) before.

Next, we discuss the implementation of these mechanisms and their integration into a realization of GMM: the AM3 tool.

4.2 Implementation

Our type system was firstly emulated using System Coq, and then prototyped as a Java stand-alone system. In such an implementation terms are constructed in a bottom-up fashion, as discussed above, and types are inferred following the type rules. Our experiments were satisfactory, as the right types where found for well-typed terms, and also ill-typed terms correctly threw exceptions during their construction. However, our goal is to integrate such an implementation into the prototypical realization of GMM, which is provided by the Eclipse-GMT AM3 project [1]. Thus, some general information about its overall architecture and main features is first presented. Then, more details on the integration of the type system into the AM3 GMM prototype are given, still taking the same *KM32ATLCopier* transformation as a test example.

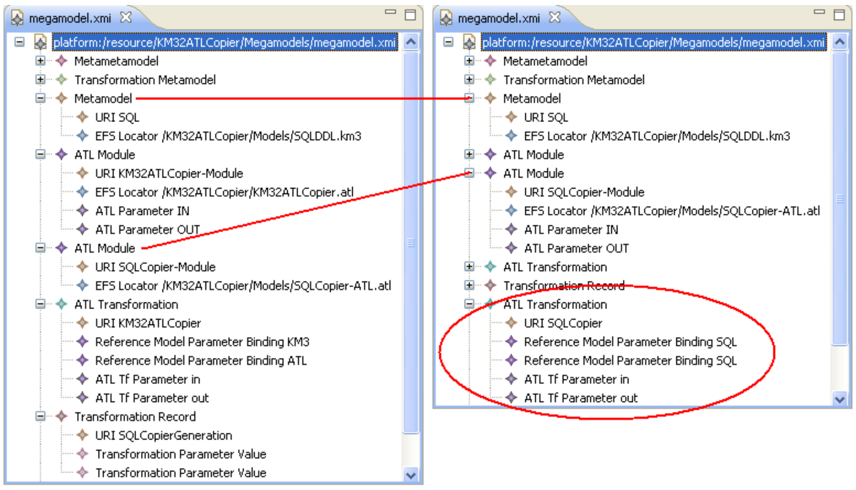
The Eclipse-GMT AM3 Global Model Management Solution. The current version of the Eclipse.org *AM3* solution implements the previously described conceptual framework and thus can be used as the GMM tool in the context of our experiments with the proposed type system. It is a project which is part of the *GMT* subproject, which is itself part of the top-level Eclipse *Modeling* project. As an Eclipse project, the AM3 prototype is fully open-source and thus all its source code is freely available from its Eclipse website and download server [1]. The generic and extensible AM3 Global Model Management solution, built on top of the Eclipse environment, provides not only the capabilities to explicitly specify the metadata associated with a given modeled system or MDE process, but also a standard *Megamodel Navigator* as well as generic and extensible editors for instantiating and editing the megamodel in a more user-friendly way. In addition, it offers several extension points allowing the definition of domain-specific extensions of the tool (i.e., extending both the metamodel of megamodel and the related UI components). Thus, AM3 is composed of two distinct sets of Eclipse plug-ins:

– The *core* plug-ins provide the basic metamodel of megamodel, the core run-time environment, the main APIs and an associated generic navigator and editors.
– The *extension* plug-ins provide extensions of the metamodel of megamodel, related specific APIs and corresponding extensions of the UI (for instance specific editor pages, contextual actions, etc).

With AM3, users can build their customized Megamodeling solution by extending either the core plug-ins or other already existing extension plug-ins. Indeed, a set of generic MDE extensions have already been developed: *GMM* for Global Model Management which implements the previously presented GMM conceptual framework, *GMM4ATL* for dealing with model transformations in ATL, *GMM4CT* for supporting Composite Transformations, etc.

Integrating the type system into the AM3 Solution. In order to fully exploit the benefits of the type system in a concrete environment, it must be integrated into the current implementation of the AM3 tool. To that end, and based on the mapping between terms and GMM constructs presented in Sect. 3.1, possible implementation steps could be the following:

1. Develop the required interfaces allowing external tools to provide/retrieve information to/from the existing implementation of the type system;
2. Extend the current *GMM* extension of the AM3 tool so that all the information needed by the type system for a successful evaluation can be represented into a megamodel;
3. Modify the current transformation executor in the *GMM4ATL* extension so that, in the case of the execution of a HOT implemented in ATL, the


megamodel.xmi

```
platform:/resource/KM32ATLCopier/Megamodels/megamodel.xmi
  Metametamodel
  Transformation Metamodel
  Metamodel
    URI SQL
    EFS Locator /KM32ATLCopier/Models/SQLDDL.km3
  ATL Module
    URI KM32ATLCopier-Module
    EFS Locator /KM32ATLCopier/KM32ATLCopier.atl
    ATL Parameter IN
    ATL Parameter OUT
  ATL Module
    URI SQLCopier-Module
    EFS Locator /KM32ATLCopier/Models/SQLCopier-ATL.atl
  ATL Transformation
    URI KM32ATLCopier
    Reference Model Parameter Binding KM3
    Reference Model Parameter Binding ATL
    ATL Tf Parameter in
    ATL Tf Parameter out
  Transformation Record
    URI SQLCopierGeneration
    Transformation Parameter Value
    Transformation Parameter Value
```

megamodel.xmi

```
platform:/resource/KM32ATLCopier/Megamodels/megamodel.xmi
  Metametamodel
  Transformation Metamodel
  Metamodel
    URI SQL
    EFS Locator /KM32ATLCopier/Models/SQLDDL.km3
  ATL Module
  ATL Module
    URI SQLCopier-Module
    EFS Locator /KM32ATLCopier/Models/SQLCopier-ATL.atl
    ATL Parameter IN
    ATL Parameter OUT
  ATL Transformation
  Transformation Record
  ATL Transformation
    URI SQLCopier
    Reference Model Parameter Binding SQL
    Reference Model Parameter Binding SQL
    ATL Tf Parameter in
    ATL Tf Parameter out
```

Fig. 3. Megamodel Samples for the *KM32ATLCopier* transformation (respectively before/after type derivation)

required information is provided to the implementation of the type system. The result of its evaluation is then retrieved, by the AM3 tool, in order to automatically fill the megamodel with the complete type information.

Additionally, corresponding editors also need to be updated. In the current implementation, terms such as *Class2Relational* from Sect. 3.1, which is a first-order atomic transformation, can be created. For the *KM32ATLCopier* case which is a HOT, specifying *KM3* as the source metamodel must introduce a variable *M*. Then, specifying *ATL* as the target metamodel should allow the user to express that *M* will be both the source and the target of the resulting transformation. In this way, all the required information for deriving an appropriate type is available.

As an illustration, let us consider a megamodel registering the KM32ATLCopier model transformation (along with the KM32ATLCopier-Module transformation model) and the SQL metamodel. After KM32ATLCopier is applied to SQL according to the current AM3 implementation, the SQLCopierGeneration transformation record is created. Its target model is therefore the SQLCopier-Module transformation model. The state of the megamodel is shown in the left part of Fig. 3. Note that the source and target models of SQLCopier-Module were not created because the current typing approach does not provide enough information for doing so. Additionally, the corresponding model transformation (i.e., SQLCopier) was not created for the same reason. The megamodel resulting from deriving the type of the result of such an execution is shown in the right part of Fig. 3. According to the type derivations discussed before, it is possible to know that the result of the execution has type $SQL \rightarrow SQL$, and thus the information for properly completing SQLCopier-Module and creating the SQLCopier relationship is available.

To summarize, the implementation of the type system is integrated into the GMM prototype in such a way that it allows automatically inferring the previously lacking type information. Thus, the corresponding megamodel can now be automatically updated with the computed information.

5 Related Work

GMM is about managing models and other MDE-related resources which are defined elsewhere. So far the only exception to this is that composite transformations can in fact be defined in GMM. Typing becomes a critical issue when execution is considered, and can be studied both at intra-resource and inter-resource levels. In the former case, typing deals with elements within a resource, and the focus is on their internal properties. For example, a type system for a transformation language could ensure that produced models will satisfy some properties [9], such as good behavior. In the latter case, elements to be typed are the resources themselves. Typing in GMM mainly takes this second form. However, well typing of composite transformations is important to us as well.

Similarly to GMM, [10] presents a metamodel for describing MDE concepts and their relationships. Unlike GMM, only core concepts are considered and no tool support is reported. In particular, the typing of those concepts is not addressed or discussed, as we did for GMM.

Model typing is addressed in [16] for investigating transformation reuse. A form of subtyping for model types (i.e., metamodels) enables a sort of *subsumption* on models. Under some circumstances the same transformation may be applied to models of different types. A basic transformation language was introduced for discussing those circumstances, and a type system was defined for it. In that language, transformations are in-place procedures rather than functions, thus they may not be composed. In addition, they are not treated as models and HOTs are not addressed. Although it is related to inter-resource issues due to the subtyping relation, that type system, compared to ours, mainly deals with internal concerns of transformation definitions.

Constructive Type Theory was used in [15] for encoding the MOF layered metamodeling architecture. In particular, an infinite hierarchy of sorts was used for that purpose. However similar, the MOF hierarchy presents an extra level compared to GMM's. Additionally, the dual representation of elements at one level as types of that level and instances of types of the level above was represented, requiring reflection maps for establishing such a correspondence. Since MOF was the only metametamodel, no additional hierarchies were required as in our case. Such a formalism focuses on MOF and therefore is closed to the representation of MOF-based artifacts, which include metamodels, models, and so on, but excludes other MDE-based artifacts. In particular, model transformations and their execution was not considered in that framework.

Typechecking of compositions of transformations has been addressed in [20] and with more detail in [18]. Both approaches use different notions of model typing, and like ours, they require the same type for connecting two adjacent subtransformations. However, none of them provide explicit rules to that end. Additionally, HOTs as well as other cases discussed in this work are not handled.

6 Conclusions and Further Work

Typing in GMM enables transformation execution and is required for preventing type errors during that execution. We improved the current typing approach by proposing a type system that formally indicates how to reason about types in GMM. We showed how non-trivial situations, such as the use of HOTs, can now be handled. Although constructs like model weaving and model-to-text/text-to-model transformations are not yet supported, the issues they pose are similar to those we already dealt with in this work.

Our type system ensures good behavior but relying on the good behavior of atomic transformations. A stronger level of type safety would be achieved by integrating our type system with the type system of a transformation language. This is delicate since both type systems need to be aligned and the result of the integration should still be sound. ATL would be an appropriate case for this. In turn, our type system would benefit from including features from $F_{2<:}$ [8]. This would enable subtyping, not only for model types as in [16], but also for function types as well. Moreover, composite transformations are currently defined by the user, and type information can be used for supporting this manual process, and even for automating (parts of) it.

Acknowledgements

We would like to thank Alvaro Tasistro for his numerous contributions to this document. The present work has been jointly supported by the IST-FP6 MODELPLEX European project [13] and the Conicyt Chile-INRIA HOT project.

References

1. AM3 Project. Internet (2009), http://www.eclipse.org/gmt/am3/
2. ATL Transformations Zoo. Internet (2009), http://www.eclipse.org/m2m/atl/atlTransformations/
3. Barbero, M., Jouault, F., Bézivin, J.: Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In: 15th ECBS 2008. IEEE, Los Alamitos (2008)
4. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
5. Bézivin, J., Jouault, F.: KM3: a DSL for Metamodel Specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)

6. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005)
7. Cardelli, L.: Typechecking Dependent Types and Subtypes. In: Boscarol, M., Levi, G., Aiello, L.C. (eds.) Foundations of Logic and Functional Programming. LNCS, vol. 306, pp. 45–57. Springer, Heidelberg (1988)
8. Cardelli, L.: Type Systems. In: Tucker, A.B. (ed.) The Computer Science and Engineering Handbook, pp. 2208–2236. CRC Press, Boca Raton (1997)
9. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal 45(3), 621–646 (2006)
10. Favre, J.-M.: Towards a Basic Theory to Model Model Driven Engineering. In: 3rd Workshop in Software Model Engineering, Lisbon, Portugal (2004)
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
12. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
13. MODELPLEX IST-FP6 European Project. Internet (2009), https://www.modelplex-ist.org/
14. Paulin-Mohring, C.: Le Système Coq. Thèse d'habilitation, ENS Lyon (1997)
15. Poernomo, I.: A Type Theoretic Framework for Formal Metamodelling. In: Reussner, R., Stafford, J.A., Szyperski, C. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 262–298. Springer, Heidelberg (2006)
16. Steel, J., Jézéquel, J.-M.: On Model Typing. Software and System Modeling 6(4), 401–413 (2007)
17. The Coq Proof Assistant Reference Manual. Version 8.2 (2009), http://coq.inria.fr/doc-eng.html
18. Vanhooff, B., Ayed, D., Baelen, S.V., Joosen, W., Berbers, Y.: UniTI: A Unified Transformation Infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)
19. Werner, B.: Une Théorie des Constructions Inductives. Thèse de doctorat, Université Paris 7 (1994)
20. Willink, E.D.: OMELET: Exploiting Meta-Models as Type Systems. In: Akehurst, D.H. (ed.) 2nd European Workshop on MDA, pp. 160–164. University of Kent (2004)
21. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Inf. Comput. 115(1), 38–94 (1994)

Supporting Parallel Updates with Bidirectional Model Transformations

Yingfei Xiong[1], Hui Song[2], Zhenjiang Hu[1,3], and Masato Takeichi[1]

[1] Department of Mathematical Informatics
University of Tokyo, Tokyo, Japan
{Yingfei_Xiong,takeichi}@mist.i.u-tokyo.ac.jp

[2] Key Laboratory of High Confidence Software Technologies (Peking University)
Ministry of Education, Beijing, China
songhui06@sei.pku.edu.cn

[3] GRACE Center
National Institute of Informatics, Tokyo, Japan
hu@nii.ac.jp

Abstract. Model-driven software development often involves several related models. When models are updated, the updates need to be propagated across all models to make them consistent. A bidirectional model transformation keeps two models consistent by updating one model in accordance with the other. However, it does not work when the two models are modified at the same time.

In this paper we propose a new algorithm that wraps any bidirectional transformation into a synchronizer with the help of a model difference approach. The synchronizer enables parallel updates by taking the two original models, the two updated models as input and producing two new models where the updates are synchronized. We also examine the requirements for synchronizing parallel updates, and demonstrate that our algorithm satisfies the requirements if the bidirectional transformation satisfies the *correctness* property and the *hippocraticness* property. Implementation of our algorithm showed that it works well in a runtime management framework in practical cases.

1 Introduction

One central activity of model-driven software development is to transform high-level models into low-level models through model transformation. For example, Figure 1(a) shows a basic Unified Modeling Language (UML) model containing a `Book` class with two attributes. To implement this UML design, we can write a model transformation program to transform the model into a basic database model, as shown in Figure 1(b). Each UML class whose `persistent` feature is true is transformed into a database table of the same name. Each attribute belonging to a persistent class is transformed into a column with the same name. The database model also contains implementation-related information, the `owner` feature, and this feature is set with default value `"admin"`.

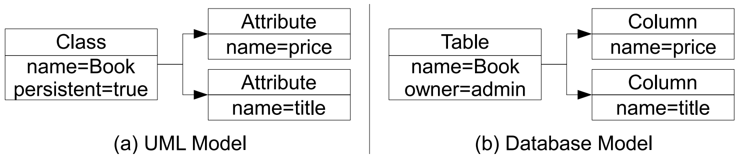


Fig. 1. Transforming a UML model into a database model

In an ideal situation, the target model is always obtained from a source model and never needs to be modified. In reality, however, developers often need to modify the target model directly. In such cases, the updates need to be reflected back to the source model.

Bidirectional model transformation [1,2] solves this maintenance problem by providing a bidirectional model transformation language, which is used to describe the relation between the two models symmetrically. Programs in these languages are used not only to transform models from one format into another, but also to update the other model automatically when a model is updated by users.

Stevens [3] formalizes a bidirectional model transformation as two functions. If M and N are meta models and $R \subseteq M \times N$ is the consistency relation to be established between them, a *bidirectional model transformation* consists of two functions:

$$\overrightarrow{R} : M \times N \to N$$
$$\overleftarrow{R} : M \times N \to M$$

Given a pair of models $(m, n) \in M \times N$, function $\overrightarrow{R}$ changes n to make it consistent with m. Similarly, $\overleftarrow{R}$ changes m in accordance with n. Many bidirectional model transformation languages fall into this model; typical languages include Query/View/Transformation relations (QVT-R) [1] and TGGs [2].

However, in some cases, models m and n may be simultaneously updated before a bidirectional transformation can be applied. For example, a designer could be working on the design model at the same time a programmer is working on the implementation model. Applying the transformation in either direction will result in the loss of updates on the target side.

Because of the large number of available bidirectional transformation languages and existing transformation programs, it would be preferable if we could synchronize parallel updates using existing bidirectional transformations. One basic idea is to sequentially apply the two updates and interleave them with two transformations. For example, suppose a user changes the `price` attribute into `"bookPrice"` in the UML model and another user changes the `title` column into `"bookTitle"` in the database model at the same time, as shown in Figure 2. We can assume that the `title` column in the database model is changed first and perform a backward transformation to change the `title` attribute in the UML model. Then, we change the `price` attribute into `"bookPrice"` in the UML model and perform a forward transformation to change the `price` column in the database model.

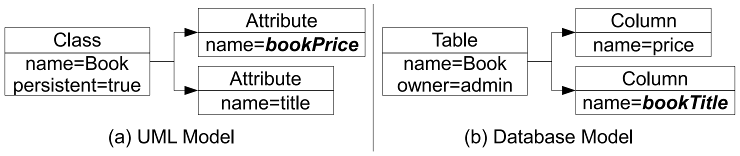


Fig. 2. Non-conflicting parallel updates

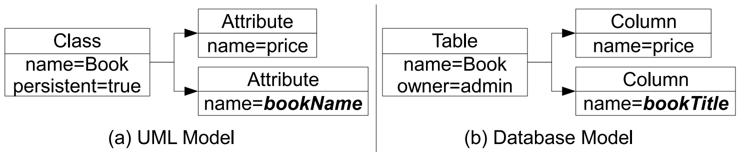


Fig. 3. Conflicting parallel updates

However, there are two problems in implementing this idea. First, as with bidirectional transformation, we do not want to require users to track updates. We thus need to identify which part of the updated UML model was changed so that we can later apply the update to the result of the backward transformation. Second, the updates applied to the two models can sometimes conflict. Figure 3 shows an example of conflicting updates where the `title` attribute and the `title` column are changed to different values. If we transform backward and then go forward again, we will lose the update to the database model. A preferable synchronization procedure would detect such conflicts and advise the user.

In this paper we propose a new approach based on the idea of sequentially applying parallel updates. We use commonly used model difference approaches [4,5,6] to solve the two problems above. We design an algorithm that use model difference approaches to wrap any bidirectional transformation into a synchronizer for parallel updates. The synchronizer takes the two original models and two updated models as input and produces two new models in which the updates are synchronized.

The main contributions of this work can be summarized as follows.

– We identify general requirements for synchronizing parallel updates. The requirements mainly consist of three properties: *consistency*, *stability* and *preservation*. These properties are adapted from previous work [7] on non-symmetrical, language-specific synchronization. We significantly modify them to make them appropriate for more general and symmetrical synchronization.
– We propose an algorithm that can wrap any bidirectional model transformation and any model difference approach into a synchronizer supporting parallel updates. It treats the bidirectional model transformation and the model difference approach as black boxes and does not require the user to write additional code. For any bidirectional transformation satisfying the *correctness* and *hippocraticness* properties [3], the synchronizer satisfies the

consistency, *stability*, and *preservation* properties, ensuring correct and predictable synchronization behavior.
- We have implemented our algorithm and applied it to a runtime management framework. The application showed that our algorithm works well in practical cases.

The rest of the paper is organized as follows. Section 2 describes the bidirectional model transformation properties introduced by Stevens [3]. Section 3 introduces our requirements for synchronizing parallel updates. Section 4 describes model difference approaches in our context and introduces how we use a model difference approach to construct a three-way merger and a preservation tester, which are used in our algorithm. Section 5 introduces our algorithm and proves that bidirectional model transformation properties lead to model synchronization properties. Section 6 describes its application and Section 7 discusses related work. Finally, Section 8 concludes the paper and discusses a possible future direction: conflict resolution.

2 Background: Properties of Bidirectional Model Transformation

The definition of bidirectional transformation describes only the input and output of a transformation; it does not constrain the behavior of the transformation. Stevens [3] proposes three properties that a bidirectional transformation should satisfy to ensure that models are transformed in a reasonable way. In this paper, however, we require only that a bidirectional transformation satisfy two of them (*correctness* and *hippocraticness*) because the last property, *undoability*, would prohibit many practical transformations.

The first property, *correctness*, ensures that a bidirectional transformation does something useful. Given two models, m and n, the forward and backward transformations must establish consistency relation R between them.

Property 1 (Correctness).

$$\forall m \in M, n \in N : \qquad R(m, \overrightarrow{R}(m, n))$$
$$\forall m \in M, n \in N : \qquad R(\overleftarrow{R}(m, n), n)$$

The second property, *hippocraticness*, prevents a bidirectional transformation from doing something harmful. Given two consistent models m and n, if neither model is modified, the forward and backward transformations should modify neither model.

Property 2 (Hippocraticness).

$$R(m, n) \Longrightarrow \overrightarrow{R}(m, n) = n$$
$$R(m, n) \Longrightarrow \overleftarrow{R}(m, n) = m$$

The last property, *undoability*, means that a performed transformation can be undone. Suppose there are two consistent models, m and n. A user, working on the M side, updates m to m' and performs a forward transformation to propagate the updates to the N side. Immediately after the transformation, he realizes that the update is a mistake. He modifies m' back to m and performs the forward transformation again. If the bidirectional transformation satisfies *undoability*, the second transformation will produce the exact n to cancel the previous modification on the N side.

Property 3 (Undoability).

$$\forall m' \in M : \quad R(m,n) \Longrightarrow \overrightarrow{R}(m, \overrightarrow{R}(m',n)) = n$$
$$\forall n' \in N : \quad R(m,n) \Longrightarrow \overleftarrow{R}(\overleftarrow{R}(m,n'),n) = m$$

While *undoability* makes sense in some situations, here we do not require bidirectional transformations to satisfy this property because *undoability* imposes a strong requirement on the consistency relation, R, and prohibits many useful transformations. One example is the UML-to-database transformation we mentioned in Section 1. If we change the `persistent` property of a class to `false` in the UML model, a forward transformation will delete the corresponding table in the database model. However, if we modify the property back to `true`, it is not possible for the forward transformation to recover the original table because the value of the `owner` property has been lost. This problem cannot be solved from the transformation alone. To satisfy *undoability*, we must change the meta model of the database to store all deleted `owner` properties, which would be impossible and unnecessary in many cases.

3 Requirements of Synchronizing Parallel Updates

As discussed above, the interface of the bidirectional transformation functions do not allow parallel updates and we need a new interface. Suppose M and N are meta models and $R \subseteq M \times N$ is the consistency relation to be established. A *synchronization procedure* for parallel updates is a partial function of the following type.

$$sync : R \times (M \times N) \rightarrow M \times N$$

This definition describes the input and output of the synchronization procedure. The input includes four models: the two original models satisfying consistency relation R, and the two updated models. The output is two new models for which the updates are synchronized.

This definition already implies some requirements for synchronizing parallel updates. First, the synchronization procedure is a function, which means that this procedure must be deterministic. Second, the function is partial, which implies detection of conflicts in updates. If the updates to the two models conflict, the function should be undefined for these input.

However, like bidirectional transformations, this definition alone does not impose much constraint on the behavior of the synchronization. We introduce three properties to ensure the synchronization procedure behaves in a reasonable way. These properties were first proposed in previous work [7], and are significantly modified for the synchronization of parallel updates.

Similar to the properties of bidirectional transformation, our first property, *consistency*[1], requires that the synchronization procedure to do something useful. It ensures that consistency relation R is established on the output models.

Property 4 (Consistency).
$\mathit{sync}(m, n, m', n')$ is defined $\Longrightarrow R(\mathit{sync}(m, n, m', n'))$

The second property, *stability*, prevents the synchronization procedure from doing something harmful. If neither of the two models has been updated, the synchronization procedure should update neither of them.

Property 5 (Stability).
$R(m, n) \Longrightarrow \mathit{sync}(m, n, m, n) = (m, n)$

The last property, *preservation*, is more interesting. Consider the updates shown in Figure 2. The easiest way to achieve consistency is to change the attribute name from `"bookPrice"` back to `"price"` and change `"bookTitle"` back to `"title"`. However, this is not the behavior we want. What we want is that the updates are propagated from the modified parts to the unmodified parts, rather than changing back the modified parts. To prevent the unwanted behavior, we require that the user updates be preserved in the output models. If the user changes the name of the price attribute to `"bookPrice"`, the synchronization procedure should not change the attribute to any other value.

Formally, let $P_M \in M \times M \times M$ be a preservation relation over M, in the sense that $P_M(m_o, m_a, m_c)$ implies that the update from m_o to m_a is preserved in m_c. Similarly, let $P_N \in N \times N \times N$ be a preservation relation over N.

Property 6 (Preservation).
$\mathit{sync}(m, n, m', n') = (m'', n'') \Longrightarrow P_M(m, m', m'')$
$\mathit{sync}(m, n, m', n') = (m'', n'') \Longrightarrow P_N(n, n', n'')$

Note that we do not define a universal preservation relation for all meta models. Instead, we allow different preservation relations to be defined for one meta model, and a synchronization procedure should satisfy a specific preservation relation. This is because there may be multiple modification sequences from one model to another, and a different choice of modification sequences leads to a different preservation result.

For example, in Figure 3(a), we change the `name` feature of the `Attribute` object from `"title"` to `"bookName"`. However, we can also consider the update as deleting the `Attribute` element `"title"` and adding a new `Attribute` element `"bookName"`. The same dilemma applies to the database model. As a result, if we adopt the feature-changing update, the updates on the two models conflict and we cannot find a consistent model that preserves both updates. However, if we adopt the object-deleting-adding update, the updates to the two models do not conflict, and the model in Figure 4 preserves the updates. As a result, the preservation relation depends on what update operations we consider and how

[1] This was called *propagation* in the previous publication [7].

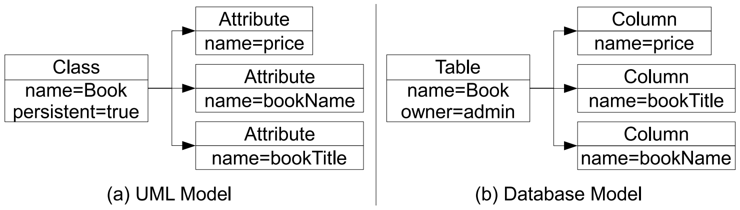


Fig. 4. Updates to both models are preserved

we recover updates from models. In the next section we will define a preservation relation from a model difference approach.

The previous work [7] also introduces a fourth property: *composability*. However, this property has the same problem as *undoability*: it constrains the consistency relation too much and prohibits many useful transformations. Therefore, we do not require the synchronization procedure to satisfy this property.

4 Model Difference, Three-Way Merger and Preservation

As introduced in Section 1, we use model difference approaches [4,5,6] to identify updates and detect conflicts. In this section we describe model differences in our context. We will also show how we use a model difference approach to define a three-way merger and a preservation relation, which will be used in our algorithm.

4.1 Model Difference

Following the definitions of Diskin [8], we consider the space of models in the meta model M as a directed graph; its nodes are models, and its arrows are updates. We call the starting node of update δ the *pre-model* of δ (denoted as $\delta.pre$) and the end node of δ the *post-model* (denoted as $\delta.post$). There may be different updates leading from one model to another, so the graph is a multigraph, meaning that there can be more than one arrow between two nodes. In addition, any model in M should be updatable to any model, so the graph is a complete graph. This definition is different from that in other work [9,10] in which updates are considered to be functions. In our definition, each update has only one associated pre-model and only one associated post-model, and cannot be directly applied to other models. We use Δ_M to denote the set of updates in the model space of M.

We consider that a model difference approach should provide at least two operations. The first operation is used to identify the updates in two models. We call it the *difference operation*. Formally, a difference operation is a function, $\mathit{diff} \in M \times M \rightarrow \Delta_M$, that takes two models, m and m', and produces update δ, where $\delta.pre = m$ and $\delta.post = m'$. We define a difference operation as a function to require the procedure to be deterministic. A difference operation should choose one update from all possible updates using predefined criteria. For example, in Alanen et al.'s approach [4], the result is a set of insertions and deletions that preserve the longest common subsequence when comparing two ordered features.

The second operation, *the union operation*, also known as "parallel composition" in some publications [9], is used to merge different updates to be applied to the same model. This operation is useful in distributed development environments where several developers may simultaneously work on the same model, and their updates need to be merged. Given updates δ_1 and δ_2 where $\delta_1.pre = \delta_2.pre$, we denote their union as $\delta_1 + \delta_2$, where $(\delta_1 + \delta_2).pre = \delta_1.pre = \delta_2.pre$ and $(\delta_1 + \delta_2).post$ is a model that is considered to have both δ_1 and δ_2 applied. The union operation should be commutative, that is, $\delta_1 + \delta_2 = \delta_2 + \delta_1$. In addition, we do not require the union operation to be total. If δ_1 and δ_2 conflict, $\delta_1 + \delta_2$ is undefined. The techniques to implement this operation can be found in existing approaches [4,9].

For example, given the model in Figure 1(a) and the model in Figure 2(a), a difference operation may return the update (intuitively) "change the `price` attribute in Figure 1(a) to `bookPrice`". Similarly, for Figure 1(a) and Figure 3(a) it may return "change the `title` attribute in Figure 1(a) to `bookName`". The union of the two updates may be a new update that changes both attributes in Figure 1(a).

One special case in the model difference function and the union operation is the identity update, which means nothing is changed. We require that the difference operation always returns the identity update when comparing two identical models and that computing the union of arbitrary update δ with the identity update results in δ. Formally, we require that the *diff* function and the "+" operator satisfy the following property.

Property 7 (Stability of Model Difference).
$\forall \delta \in \Delta_M : \quad \delta + \mathit{diff}(\delta.pre, \delta.pre) = \delta$

4.2 Three-Way Merger

With the model difference function and the union operator, we can construct a three-way merger of models. A *three-way merger* takes one original model and two independently updated copies of the model and produces a new model in which the updates to the two copies are merged. Three-way mergers are widely used in many distributed systems, like the Concurrent Versions System (CVS), and in the `diff3` command [11] in Unix. Given an original model m_o and two independently modified copies, m_a and m_b, a three-way merger is a partial function defined as the following.

$$\mathit{merge}(m_o, m_a, m_b) = (\mathit{diff}(m_o, m_a) + \mathit{diff}(m_o, m_b)).post$$

If $(\mathit{diff}(m_o, m_a) + \mathit{diff}(m_o, m_b))$ is not defined, *merge* is not defined, indicting there are conflicts between m_a and m_b.

4.3 Preservation

In Section 3 we have mentioned that there are multiple preservation relations for one meta model if there are multiple updates from a pair of models. As model difference approaches identify an update using certain criteria, we can define a preservation relation in accordance with the semantics of a model difference approach.

Definition 1. Given a difference operation *diff* and a union operator "+", we say m_c *preserves* the update from m_o to m_a if and only if there exists an update δ where $(\mathit{diff}(m_o, m_a) + \delta).\mathit{post} = m_c$.

One natural result is that a three-way merger will always preserve the updates in both models.

Theorem 1. *If* $m_c = \mathit{merge}(m_o, m_a, m_b)$*, then* m_c *preserves the update from* m_o *to* m_a *and the update from* m_o *to* m_b*.*

Proof. From the definition of *merge* we get $(\mathit{diff}(m_o, m_a) + \mathit{diff}(m_o, m_b)).\mathit{post} = m_c$. From the commutativity of +, we get $(\mathit{diff}(m_o, m_b) + \mathit{diff}(m_o, m_a)).\mathit{post} = m_c$. Because there exists $\mathit{diff}(m_o, m_b)$, from the first formula, we have that m_c preserves the update from m_o to m_a. Similarly, from the second formula, we have that m_c preserves the update from m_o to m_b.

This definition of preservation gives us a basic method for testing whether three models (m_o, m_a, and m_c) satisfy the preservation relation. However, to actually test it, we must iterate all possible updates starting from m_o, which is not possible in practice. What we need is an efficient procedure for quickly testing the preservation of three models. Such an efficient testing procedure is difficult to find in general. However, given a specific model difference approach, it is often possible to define an efficient testing procedure in accordance with the update operations considered in the difference approach. In the following we show how to efficiently test preservation for Alanen et al.'s [4] model difference approach as an example.

Testing Preservation in Alanen et al.'s Approach. Alanen et al. consider an update as a sequence of update operations, and they define seven types of operations, as shown in Table 1. In their work, they assume that each element has a universally unique identifier (UUID) that does not change across versions. Under this assumption, we can easily identify and match model elements in different versions of objects. In addition, they consider limited types of features on the models. Features can be classified as attributes that store primitive values and references that store links to other model elements. They assume that all attributes are single features (can contain only one value) and that all references are multiple features (can contain more than one feature, either ordered or unordered).

To test whether an update from m_o to m_a is preserved in m_c, we first use the difference operation to get the update $\delta_{oa} = \mathit{diff}(m_o, m_a)$. Then we examine

Table 1. Modification Operations

Operation	Description
new(e, t)	create a new element e of type t
delete(e, t)	delete element e of type t
set(e, f, v_o, v_n)	set an attribute f of element e from v_o to v_n
insert(e, f, e_t)	add a link from $e.f$ to e_t for an unordered reference f
remove(e, f, e_t)	remove a link from $e.f$ to e_t for an unordered reference f
insertAt(e, f, e_t, i)	add a link from $e.f$ to e_t at index i for an ordered reference f
removeAt(e, f, e_t, i)	remove a link from $e.f$ to e_t at index i for an ordered reference f

Table 2. Testing of Preservation

Operation in δ_{oa}	Preservation condition
new(e, t)	e exists in m_c, and all features of e are the same as m_a
delete(e, t)	e does not exist in m_c
set(e, f, v_o, v_n)	e exists in m_c, and $e.f$ is the same value as v_n
insert(e, f, e_t)	e exists in m_c, and a link to e_t exists in $e.f$
remove(e, f, e_t)	e does not exist in m_c, or a link to e_t does not exist in $e.f$
insertAt(e, f, e_t, i)	e exists in m_c, a link to e_t exists in $e.f$, and the inserted links have their order in m_a preserved in m_c for all insertAt operations on the feature
removeAt(e, f, e_t, i)	always preserved (as deleted links can be inserted back)

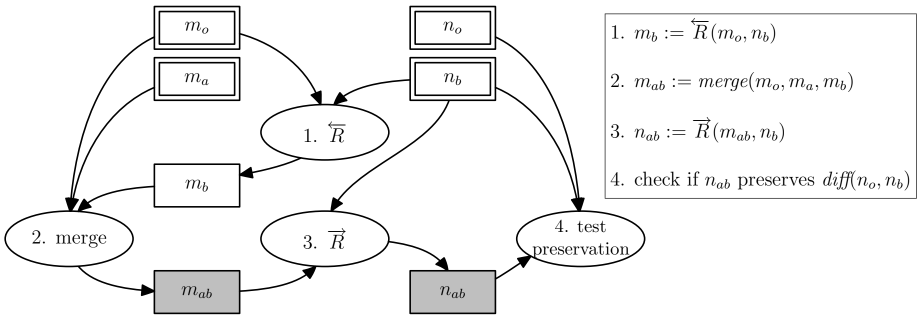
m_c for each update operation in δ_{oa}. If we find that an operation such that the union of any operation and this operation cannot reach m_c from m_o, we report a violation of preservation. The detailed rules for examining the update operations can be found in Table 2.

For example, suppose the `price` attribute in Figure 1(a), the `bookPrice` attribute in Figure 2(a), and the `price` attribute in Figure 3(a) share UUID e_p. The difference of Figure 1(a) and Figure 2(a) is thus an update containing one update operation: set(e_p, `name`, `"price"`, `"bookPrice"`). This update is not preserved in Figure 3(a) because the rule for `set`(e, f, v_o, v_n) is violated: e_p.`name` has a value of `"price"` and is different from `"bookPrice"` in Figure 3(a).

5 Algorithm

Now we have a three-way merger and can test the preservation of updates. Let us use them to wrap a bidirectional transformation into a synchronizer for parallel updates. The basic idea is to first convert the model from the N side to the M side using backward transformation, then use the three-way merger to reconcile the updates, and transform back using the forward transformation. The detailed algorithm is shown in Figure 5.

We explain the algorithm using the example in Section 1. Initially, we have the two models in Figure 1, which correspond to m_o and n_o in our algorithm. Users modify the two models into the models in Figure 2, which correspond to m_a



m_o
m_a
n_o
n_b
1. $\overleftarrow{R}$
m_b
2. merge
3. $\overrightarrow{R}$
4. test preservation
m_{ab}
n_{ab}
1. $m_b := \overleftarrow{R}(m_o, n_b)$
2. $m_{ab} := merge(m_o, m_a, m_b)$
3. $n_{ab} := \overrightarrow{R}(m_{ab}, n_b)$
4. check if n_{ab} preserves $diff(n_o, n_b)$

Fig. 5. Synchronization algorithm

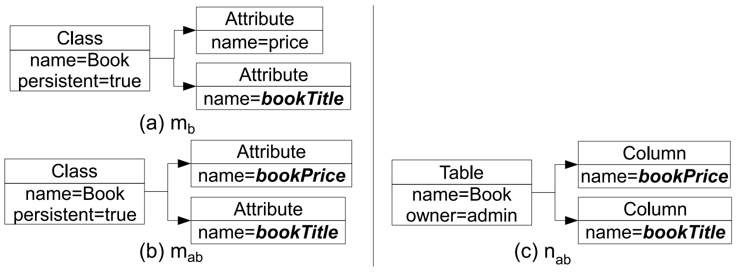


Fig. 6. Execution of algorithm

and n_b in our algorithm. We use different subscripts to show different updates, where a represents the update on m_o and b represents the update on n_o. The four models together comprise the algorithm input.

The first step of our algorithm is to invoke backward transformation $\overleftarrow{R}$ to propagate the updates made to n_b to m_o, resulting in m_b. The result is shown in Figure 6(a). The attribute name is changed from `"title"` to `"bookTitle"`.

Now we have model m_a containing update a and model m_b containing update b. The second step is to use the three-way merger we constructed in the last section to merge the two updates and produce synchronized model m_{ab} on the M side. The result is shown in Figure 6(b). The model has both attributes changed; i.e., it contains updates from both sides. If the updates to the two models conflict, the three-way merger detects the conflict and reports an error.

The third step is to use forward transformation $\overrightarrow{R}$ to produce synchronized model n_{ab} on the N side. The result is shown in Figure 6(c). This model also contains updates from both sides, with both columns changed.

Now we have two synchronized models to which the updates have propagated. It looks as if we have performed enough steps to finish the algorithm. However, the above steps do not ensure the detection of all conflicts and may lead to violation of *preservation* due to the heterogeneousness of the two models.

To see how this can happen, let us consider the example in Figure 7. Initially we have only one class and one table, and they are consistent. Then suppose that a user changes the `persistent` feature of the class to `false` and changes the owner of the table to `"xiong"`. Because the `owner` feature is not related to the UML model, the backward transformation changes nothing, and m_b is the same as m_o. The three-way merger detects no updates in m_b and produces a model that is the same as m_a. Finally, we perform the forward transformation, and the table is deleted because of the change to the `persistent` feature. However, as the user has modified a feature of the table, so he or she will expect to see the existence of the table in the final result. The input models contain conflicting updates, but the synchronization process does not detect them.

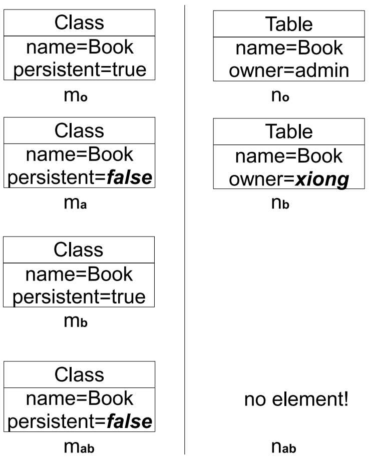


Fig. 7. Violating *preservation*

This kind of violation is caused by the heterogeneity of M and N. Due to the heterogeneity, not all updates to N are visible on the M side. As the three-way merger only works on the M side, it cannot detect such invisible conflicts.

To capture such conflict, we add an additional step, preservation testing, to the end of the algorithm. It is shown as the fourth step in Figure 5. This step uses the preservation testing procedure described in Section 4 and checks whether the update from n_o to n_b is preserved in n_{ab}. If not, the algorithm reports an error.

The models used in Figure 6 and Figure 7 are simply examples. The actual execution depends on the bidirectional transformation and the model difference approach used in the synchronization and may differ from the above execution. Nevertheless, whatever bidirectional transformation and model difference approach we choose, our algorithm ensures the three synchronization properties: *consistency*, *stability*, and *preservation*.

Theorem 2. *If the bidirectional transformation satisfies correctness, the synchronization algorithm satisfies consistency.*

Proof. Because $\overrightarrow{R}(m_{ab}, n_b) = n_{ab}$, we have $R(m_{ab}, n_{ab})$.

Theorem 3. *If the bidirectional transformation satisfies hippocraticness and the model difference approach satisfies stability of model difference, the synchronization algorithm satisfies stability.*

Proof. If we have $m_o = m_a$ and $n_o = n_b$, we have $R(m_o, n_b)$. Because of *hippocraticness*, we have $m_b = \overleftarrow{R}(m_o, n_b) = m_o$. Because of *stability of model difference*, $m_{ab} = merge(m_o, m_a, m_b) = (\mathit{diff}(m_o, m_a) + \mathit{diff}(m_o, m_b)).post = (\mathit{diff}(m_o, m_o) + \mathit{diff}(m_o, m_o)).post = m_o$. On the other hand, $n_{ab} = \overrightarrow{R}(m_{ab}, n_b) = \overrightarrow{R}(m_o, n_o) = n_o$, and the preservation testing always passes because of the existence of identity update.

Theorem 4. *The synchronization algorithm always satisfies preservation.*

Proof. Because of Theorem [1], the update on the M side is preserved. Because of the last preservation test, the update on the N side is preserved.

It is worth noting that our algorithm works even if the bidirectional transformation does not satisfy *correctness* or *hippocraticness*. This has practical value because many bidirectional transformation languages in practice do not guarantee the properties [3]. In such cases, the algorithm still produces output but does not guarantee the corresponding synchronization properties (*consistency* or *stability*).

Bidirectional transformations are symmetrical, so we can also implement this algorithm in the opposite direction. We can start a forward transformation first, merge models on the N side, perform a backward transformation, and check preservation on the M side. Implementing the algorithm in both directions can guarantee the three properties. However, due to the heterogeneity of M and N, it is possible that different directions may produce different results for some input. The difference is related to the specific bidirectional transformation approach and the difference approach used in the algorithm, and we do not discuss it in this paper.

6 Application

We implemented our algorithm in a runtime management framework [12], as shown in Figure [8]. We used our algorithm to wrap a QVT-R program [1] (executed in mediniQVT [13]) and a Beanbag-based model difference approach [10] into a synchronizer for parallel updates, and used our synchronizer to synchronize a runtime management user interface and a running system.

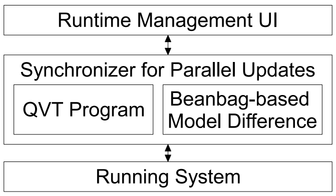

Runtime Management UI
Synchronizer for Parallel Updates
QVT Program
Beanbag-based Model Difference
Running System

Fig. 8. Structure of runtime management system

A high-level management user interface (UI) is often provided in a runtime management system for monitoring the state of the running system and for reconfiguring it. Because the high-level management UI often abstracts away many low-level details, the high-level UI and the running system are heterogeneous and need to be synchronized. Because the system state is constantly changing during runtime, any modification to the management UI will cause parallel updates. In our implementation, we captured both the management UI and the running system as models and used our algorithm to synchronize them.

The bidirectional transformation used to synchronize the two models is a QVT-R program. The QVT-R language [1] enables rapid development of bidirectional transformations. However, it does not always guarantee *correctness* and *hippocraticness*. If a program has complex interaction with the constraints on the meta models, it may produce inconsistent result. In our implementation, we manually check the consistency of our program and the constraints on the meta models to ensure *correctness* and *hippocraticness*.

The model difference approach we used is extracted from the Beanbag system [10]. We first convert models into the Beanbag data types and then use operations provided by Beanbag to merge the models. In the conversion, we assume each model element has a unique identifier and do not consider ordered multiple features. The situation is simpler than those considered by most model difference approaches, but it is sufficient for models in runtime management. The details of the Beanbag data types and model conversion can be found in the technical report of Beanbag [10].

When there is a conflict between updates to the running system and those to the management UI, our synchronization algorithm reports an error and halts. The user needs to manually resolve the conflict and resynchronize again. However, as the system is constantly changing, it is often impossible for users to resolve all conflicts. We solve this problem by giving precedence to the updates made to the management UI. In a runtime management system, updates to the management UI are in fact control operations that the user want to perform on the system, so it is always safe to overwrite an update made to the running system with one made to the management UI. To implement this, we change the difference algorithm so that it overwrites an update made to the running system with one made to the management UI if the two updates conflict. In addition, we remove the final preservation test.

We performed a set of experiments using our runtime management framework, and the results showed that our algorithm works well. The details of the runtime management framework and the experiments can be found else where [12].

7 Related Work

Several other approaches also target synchronizing parallel updates on heterogeneous data. Typical ones include Harmony [14] and Beanbag [10].

The goal of Harmony is similar to ours: use bidirectional transformations to construct synchronizers for parallel updates. Compared to our approach, Harmony uses an asymmetrical form of bidirectional transformation, where the target is an abstract of the source. Users must design a common replica and write two transformation programs to map the replicas to be synchronized to the common replica. Our approach does not require users to design an extra model, so users can better reuse existing transformation programs. In addition, we adopt the symmetrical form of bidirectional transformation, which is more frequently used in the model transformation community.

Beanbag is a general language for synchronizing parallel updates. Different from this paper, Beanbag uses an operation-based approach: users need to tell the synchronizer what update operations have been applied, and the synchronizer returns more update operations to make the data consistent. The approach in this paper is state-based: whole copies of models (the current states of models) are taken as input and new copies of these models are returned.

Another related branch of research is detecting and fixing inconsistencies in models [15,16]. The methods developed can also be used to synchronize parallel updates but from a different perspective: only the updated models are examined, and the inconsistencies are resolved by human intervention or heuristic rules. This is very different from our objective of fully automatic, predictable synchronization behavior. Compared to them, our approach is fully automatic, and the synchronization behavior is predicable through the three properties.

Some researchers build frameworks for classifying synchronization approaches. Antkiewicz and Czarnecki [17] classifies synchronization approaches using different design decisions. Under their classification schema, our synchronization algorithm can be classified as a "bidirectional, non-incremental, and many-to-many synchronizer using artifact translation, homogeneous artifact comparison, and reconciliation with choice". Diskin [8] builds a more formal framework for bidirectional model synchronization, in which bidirectional transformation is classified into lenses, di-systems, and tri-systems on the basis of the relation between models and the number of input models. Our definition of a synchronizer for parallel updates can be considered a supplement to his framework, where we add quadruple-systems, in the sense that our synchronizer takes four models as input.

8 Conclusion and Future work

In this paper we have proposed an approach that wraps a bidirectional transformation program and a model difference approach into a synchronizer for parallel updates. Our approach is general and predictable. It is general in the sense that it allows the use of any bidirectional transformation and any model difference approach, and it is predictable because it satisfies three model synchronization properties: *consistency*, *stability* and *preservation*.

Currently, our approach only reports the existence of conflicts; it does not support conflict resolution. A preferable synchronization procedure would report the features and model elements involved in the conflicts and give a list of solutions for the user to choose from. However, such a resolution procedure is difficult to define in general because the reason for a conflict is related to the specific bidirectional transformation and the model difference approach used. We plan to design a resolution procedure based on a specific transformation language and a specific model difference approach. One idea is to use QVT-R as the transformation language and exploit the trace information recorded by QVT-R. This remains for future work.

References

1. Object Management Group: MOF query / views / transformations specification 1.0 (2008), http://www.omg.org/docs/formal/08-04-03.pdf
2. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Proc. of the 4th International Conference on Graph Transformation, pp. 411–425 (2008)

3. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
4. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
5. Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design. In: Proc. 20th ASE, pp. 204–213 (2005)
6. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D.: Differencing and merging of architectural views. In: Proc. 21st ASE, pp. 47–58 (2006)
7. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Proc. 22nd ASE, pp. 164–173 (2007)
8. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 21–36. Springer, Heidelberg (2008)
9. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: Managing model conflicts in distributed development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
10. Xiong, Y., Hu, Z., Zhao, H., Takeichi, M., Hui, S., Mei, H.: Beanbag: Operation-based synchronization with intra-relations. Technical Report GRACE-TR-2008-04, GRACE Center, National Institute of Informatics, Japan (December 2008)
11. Khanna, S., Kunal, K., Pierce, B.C.: A formal investigation of diff3. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 485–496. Springer, Heidelberg (2007)
12. Song, H., Xiong, Y., Hu, Z., Huang, G., Mei, H.: A model-driven framework for constructing runtime architecture infrastructures. Technical Report GRACE-TR-2008-05, GRACE Center, National Institute of Informatics, Japan (December 2008)
13. ikv++ technologies: medini QVT homepage, http://projects.ikv.de/qvt
14. Pierce, B.C., Schmitt, A., Greenwald, M.B.: Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania (2003)
15. Egyed, A.: Fixing inconsistencies in UML design models. In: Proc. 29th ICSE, pp. 292–301 (2007)
16. Kolovos, D., Paige, R., Polack, F.: Detecting and repairing inconsistencies across heterogeneous models. In: ICST 2008: Proceedings of the International Conference on Software Testing, Verification, and Validation, pp. 356–364 (2008)
17. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 3–46. Springer, Heidelberg (2008)

Experiments with a High-Level Navigation Language

Jesús Sánchez Cuadrado[1], Frédéric Jouault[2],
Jesús García Molina[1], and Jean Bézivin[2]

[1] Universidad de Murcia
{jesusc,jmolina}@um.es
[2] AtlanMod team, INRIA & EMN
{jean.bezivin,frederic.jouault}@inria.fr

Abstract. Writing navigation expressions is an important part of the task of developing a model transformation definition. When navigation is complex and the size of source models is significant, performance issues cannot be neglected. Model transformation languages often implement some variants of OCL as their navigation language. Writing efficient code in OCL is usually a difficult task because of the nature of the language and the lack of optimizing OCL compilers. Moreover, optimizations generally reduce readability.

An approach to tackle this issue is to raise the level of abstraction of the navigation language. We propose to complement the regular navigation language of model transformation languages with a high-level navigation language, in order to improve both performance and readability. This paper reports on the initial results of our experiments creating the HLN language: a declarative high-level navigation language. We will motivate the problem, and will we describe the language as well as the main design guidelines.

1 Introduction

Model transformations are a key element for the success of Model Driven Engineering (MDE). As this discipline becomes mature, model transformations are being used to address problems of an increasing complexity, and the number of developers writing transformations is also growing. In this way, MDE is being applied to contexts such as DSL-based development, system modernization, or megamodeling. In some scenarios (e.g. system modernization), models being handled are typically large, and performance becomes an important concern.

Model transformation languages usually rely on query or navigation languages for traversing source models in order to feed transformation rules (e.g., checking a rule filter) with the required model elements. In complex transformation definitions a significant part of transformation logic is devoted to model navigation, and most of the transformation bottlenecks are located there. In this setting, performance cannot be neglected when writing navigation expressions. However, writing efficient code can be a difficult issue, and it often compromises readability.

To tackle this issue we propose to raise the abstraction level of the navigation language, so both readability and performance may be achieved at the same time. Readability is improved as language constructs are declarative and reflect better the intention of the developer, whereas compiler optimizations are easier to perform because the granularity of the constructs is coarser. Such a high-level navigation language is intended to complement the regular navigation language implemented as part of a model transformation language.

This paper reports on the initial results of our experiments with a high-level navigation language, called HLN. We have implemented the language on top of the ATL virtual machine [2] in order to interoperate with any language implemented on top of it (e.g., ATL, QVT-R). However, HLN could also be implemented on a different engine architecture. We will explain the structure and the design of the language, and we will show why it is a good complement for navigation languages.

The paper is organized as follows. Next section motivates our approach. Section 3 presents the technical context of this work. Section 4 presents the HLN language and describes how it has been designed. Finally, Section 5 presents some related works, and Section 6 gives the conclusions.

2 Motivation

Many current model transformation languages, such as ATL [8], QVT [12], and ETL [9], use some variants of the Object Constraint Language (OCL) [15] as their navigation language. OCL encourages a "functional" style based on collections, iterators, and expressions without side-effects (e.g., collections are immutable). However, despite its apparent "declarativeness", it can be considered as a low-level navigation language since all details about the navigation steps must be specified [14]. We argue that the level of abstraction of OCL is sometimes inadequate to achieve performance.

From our experience in discovering performance patterns for model navigation in model transformation [5], we have identified four issues with OCL:

- **Algorithm locality.** Algorithms are typically defined in operations attached to metaclasses. Several identical passes are often done, computing the same value several times. Writing global algorithms as global functions often improves performance.
- **Immutability.** The lack of side-effects makes optimizing compilers very important. OCL makes intensive use of collections, and rewriting inmutable operations to mutable operations when possible is a must to get performance. However, few optimizing compilers for OCL are available.
- **Verbosity.** Efficient code tends to be large and verbose.
- **Specificity.** Similar specific algorithms to solve a given navigation problem are generally implemented several times in different transformations definitions, generally at least once per different metamodel. OCL does not provide mechanisms for generalizing algorithms.

To tackle these issues we have built a high-level navigation language, called HLN. This is a declarative language, in which each language construct is intended to address a recurrent navigation problem (i.e., a navigation pattern).

As we will show during the paper, our approach has four main advantages involving performance and readability, namely:

1. **Optimization opportunities.** Possibility for the compiler to optimize, since the language constructs are declarative and their granularity is coarse (e.g., the compiler can use mutable operations internally easily).
2. **Simplicity.** The developer does not need to know how to implement the navigation expressions in an efficient way, the best algorithm is chosen by the compiler.
3. **Readability** is improved because each construct of the language reflects exactly the intention of the developer.
4. **Generality.** Code repetition is alleviated (e.g., writing a similar algorithm for different meta-types) because of the generality of the constructs.

3 Technical Context

The technical context of our work is the ATL Virtual Machine (ATL VM) architecture provided by the AmmA platform. The ATL VM language is a small imperative instruction set composed of four categories of bytecodes: stack, memory, control flow, and model handling. It provides facilities to attach functions to metamodel elements, for instance to create helpers, as specified by OCL [15].

In this way, a language such as ATL is built by creating a compiler targeting the VM. An important benefit of targeting a VM architecture is that of language interoperability: operations defined in one language may be used from another. For instance, a library written in HLN can be reused in transformations written in several languages (provided they are implemented on top of the ATL VM).

As we will see, each HLN construct will implicitly yield to the creation of one or more helpers (i.e., operations attached to metaclasses). Such helpers can be called by any other language implemented on top of the VM (e.g., ATL), but also HLN interoperates via helpers with such languages. We do not intend to extend OCL, but to complement it "externally" with navigation libraries.

4 The HLN Language

In this section we present the High-Level Navigation (HLN) language. It is a domain-specific language for the domain of model navigation. HLN is intended to allow model transformation developers to specify model navigation statements at a high level of abstraction.

HLN is not a general purpose navigation language in the sense that it is not possible to specify all kinds of navigation on a source model. It is intended to cover a range of common navigation problems for which it provides good performance. Other problems can be solved using the other navigation language that HLN complements (e.g., OCL as generally used for model transformation). As explained above, targeting the ATL VM allows us to fulfil this requirement. The four principles guiding the design of the language are the following:

– **Declarative.** It should be based on declarative constructs. Any construct should allow to specify the result of the navigation without detailing each navigation step. Thus, the language is designed to require the minimum possible amount of information from the developer.
– **Readable.** The syntax of the constructs should resemble natural language as far a possible. Readability and maintainability are promoted since each statement reflects an intention of the developer.
– **Useful.** Each construct of the language should be included only on the basis of experimental tests showing that it provides an improvement in performance or readability.
– **Simple.** There is no "expression language" in order to keep the language simple. We rely on the interoperability with other languages (via helpers) when conditions must be expressed.

The current version of HLN comes from our experience developing a catalog of navigation patterns, which identifies recurrent problems in model navigation [1] [5]. Some of these patterns are amenable to be encoded as language constructs. In particular, we have currently implemented four of them as HLN constructs, namely: *linking* elements by some property, computing *transitive closures*, computing the *opposite* of a reference, and setting a *navigation path*.

An HLN library is composed of a header and a set of navigation statements. An excerpt of an HLN library to navigate class diagrams is shown below. The header of a library declares the source models with their reference models (i.e., metamodels) to be navigated (in the example the *CD* metamodel stands for class diagram). Then, one or more navigation statements are written, for instance, to compute the transitive closure of the *superclasses* relationship. A metamodel element is specified by prefixing the metaclass name with the corresponding metamodel name, using **!** as a separator (e.g., *CD!Class* for metaclass *Class* of metamodel *CD*). A metaclass property (possibly a helper) is specified using the dot notation (e.g., *CD!Class.superclasses*).

```
navigate IN : CD;
transitive closure allSuperclasses of CD!Class.superclasses
```

Figure 1 shows the abstract syntax of the language. The constructs inherit from the *Statement* abstract metaclass. The *HelperRef* metaclass represents a property or helper of some metaclass (note that it references *MetaElement*), but it does not say whether the property or helper exists or whether it will be created. This is part of each construct's semantics. The concrete syntax of HLN (used in examples) is textual, and is implemented with the TCS tool [7].

It is worth noting that the current implementation of the HLN compiler does not perform any static checking against the source metamodel. This means that, for instance, it is required to explicitly declare whether a property is

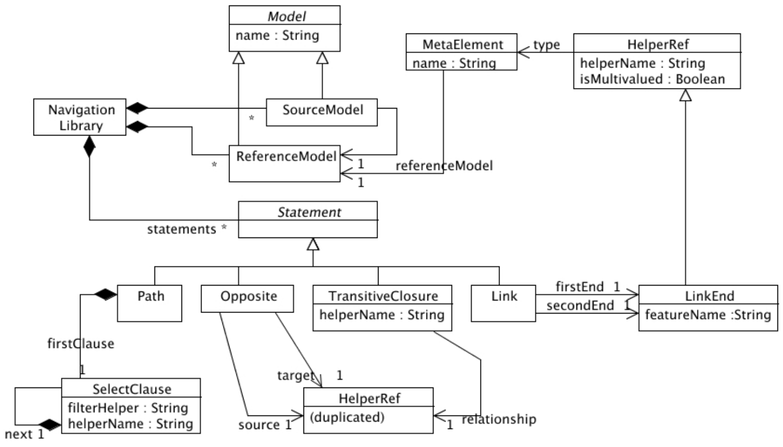


Fig. 1. Abstract syntax metamodel of HLN

multi-valued (*multivalued* attribute of *HelperRef*), despite this information being already present in the source metamodel. At the concrete syntax level this is expressed with `[*]`. The *SelectClause* and *TransitiveClosure* do not use *HelperRef*, but just *helperName*, to reference a helper to be created, because the meta-element and multiplicity are implicit.

Next, each construct will be introduced by showing a piece of simple OCL code illustrating the navigation pattern. From such code, the essential parts will be identified, and an HLN construct will be derived from them.

4.1 Linking

The *linking* pattern appears when two model elements are implicitly linked because they both have some feature (possibly helpers) with the same value (i.e., this is a kind of *join*).

For instance, a class diagram can be annotated using some weaving infrastructure such as AMW. Gathering the annotations for a given class could be done in the following way. For each class, all annotations are iterated, looking for those “pointing” to the current class.

```
context CD::Class def: annotations : Sequence(AMW::ClassAnnotation) =
AMW::ClassAnnotation.allInstances()->select(a| self.__xmiID__= a.ref)
```

Taking into account the piece of code above we derive an HLN construct, which is typically implemented using some kind of hash join. The essential parts are the following: the two metaclasses whose instances will be matched (e.g., *CD::Class* and *AMW::ClassAnnotation*), the name of the feature of each metaclass to be compared (e.g., *__xmiID__* and *ref*), and the name of helpers to be created (e.g., only *annotations* in this case). In general, two helpers will be created, one for each linked end.

The piece of abstract syntax metamodel for this construct is shown in Figure 1 (*Link* metaclass). A linking is specified as a *link* between two *link ends*. Each link end represents a metaclass and the property used to compare. A new helper containing the result of linking one end with the other end is created. In this way, we focus on the "what", and the "how" is left to the HLN's compiler.

A piece of HLN code illustrating the concrete syntax of this construct is shown below. It is intended to be read as if it were natural language: *link each Class using __xmiID__ to each ClassAnnotation using ref*). The result is that a helper called *annotations* is attached to metaclass *Class*. The _ symbol is used to indicate that we do not want to create the helper at the other end.

```
link  CD!Class.annotations[*] on __xmiID__
   to AMW!ClassAnnotation._[*] on ref;
```

Notice that the `[*]` modifier could be removed, but instead of computing a collection of links for each instance, only one link would be selected (if there are more links they are just discarded).

4.2 Navigation Path

Navigation of models with OCL is based on the dot notation to access model element properties. Collections are typically handled using iterators, such as *select* to filter elements, *collect* for deriving a collection from another one or *any* for getting an element satisfying a condition.

A typical pattern in OCL is navigating through multiple multi-valued references, filtering by some criteria, so that a combination of *select*, *collect* and *flatten* operations are needed to get the final collection. The following piece of OCL code shows this pattern in the case of obtaining all attributes contained by the classes of a package.

```
context CD::Package def : getAllAttributes : CD::Attribute =
   self.classifiers->select(c | c.isClass)->
                     collect(c | c.features)->flatten()->
                     select(f | f.isAttribute)
```

The evaluation of this expression implies creating a collection with all classifiers satisfying the *isClass* condition, next a collection containing collections of features is created, which is then flattened, and it is filtered again to obtain the result. As the number of navigation steps grow the evaluation is more inefficient.

This shows that several implementation-level details must be specified in OCL, in particular the *collect* and *flatten* operations are needed only to "normalize" the filtered collection before applying the next filter (i.e., the second *select* operation). The essential information includes navigated properties (e.g., *classifiers* and *features*) and filter conditions (if any).

The piece of metamodel for the corresponding construct is shown in Figure 1 (*Path* metaclass). An expression is based on nested navigation clauses that take the result of the owning clause to perform its navigation step. An example of the concrete syntax is shown below. The *when* part is optional if no filter is specified. This expression can be naturally read as: *given a package, select all classifiers* *satisfying the isClass condition, and, for each one, select all features satisfying the isAttribute condition.*

```
path ClassM!Package.getAllAttributes
     select classifiers  when isClass
     select features     when isAttribute
```

4.3 Opposite

Given a relationship from one model element to another, it is often necessary to navigate through the opposite relationship. For instance, in a class diagram, the opposite for the *superclasses* relationship of a *Class* metaclass is the collection of direct subclasses for a given class.

If the opposite relationship has been defined in the metamodel, then navigation in both directions can be efficiently achieved. However, such opposite relationship is not always available, so an algorithm has to be worked out.

A straightforward algorithm will involve traversing all the instances of the opposite relationship's metaclass and checking which of them are part of the relationship. For instance, to get the owning package of a class, the opposite of the package's *classifiers* relationship is computed as follows[1]:

```
context CD::Class def: owner: CD::Package =
  CD::Package.allInstances()->any(p |
     p.classifiers.includes(self) )
```

The corresponding HLN construct can be easily derived from this algorithm. The required elements are two metaclass/relationship pairs: the source metaclass and the already existing relationship, and the target metaclass with the name of the new relationship to be computed. The piece of metamodel corresponding to this construct is shown in Figure 1 (*Opposite* metaclass).

An example of the concrete syntax is shown below. It should be read in the following way: *compute the opposite relationship, called CD!Class.owner, of the relationship CD!Package.classifiers.*

```
      opposite CD!Class.owner of CD!Package.classifiers[*];
```

4.4 Transitive Closure

Computing the transitive closure of a relationship is a common operation in model transformations. An example is computing the set of all direct and indirect superclasses of a class. Another example is computing the set of reachable states from a given state of a state machine.

Let us consider a piece of OCL code to compute the transitive closure of the `superclasses` relationship in a class diagram. There are several performance issues in this code. Firstly, the `union` operation is immutable, which means that collections are duplicated unless the OCL compiler is able to detect and optimize

[1] MOF and Ecore provide the refImmediateComposite() and eContainer() operations respectively to get an element's container. However, the discussion still holds for non-containment relationships, and when such operations are not made available by the underlying transformation language.

this case. Secondly, *collect* and *flatten* also imply duplicating collections. Finally, the transitive closure is computed several times for the same class. A transitive closure can be implemented in one traversal, whereas a straightforward OCL implementation such as this one performs several.

```
context CD::Class def : allSuperclasses : Sequence(CD::Class) =
   self.parents->union(self.superclasses->
                                        collect(c | c.allSuperclasses)->flatten())
```

The only essential information to derive the HLN construct is the name of the relationship (e.g., *superclasses*), the corresponding metaclass (e.g., *CD!Class*), and the attribute helper that will be created (e.g., *allSuperclasses*). The piece of metamodel corresponding to this construct shows that (Figure 1, *TransitiveClosure* metaclass). The details about how to perform the computation are ignored This means that the compiler may evolve to implement a more efficient version of the construct, without affecting existing HLN libraries.

The concrete syntax for this construct is the following. Notice that it is implicit that the *allSuperclasses* helper must belong to *CD!Class*, and that it is multivalued.

```
transitive closure allSuperclasses of CD!Class.superclasses
```

4.5 Combining Constructs

To cover a wider range of navigation problems, while keeping the language simple, HLN allows constructs to be combined using the helpers created as a result of one construct in another construct. Again, we rely on the use of helpers to interoperate, in this case for the interoperability of the language constructs. Notice that the order of the constructs is not important, because the compiler may keep track of the dependencies.

For instance, the *transitive closure* construct does not consider a relationship defined by means of an intermediate class, such as is the case of the *superclasses* relationship in UML 2.0, which is defined using the *Generalization* metaclass.

Instead of extending HLN, the *path* construct can be used to first get the collection of direct superclasses, going through the *generalization* relationship. Then, the helper created for this construct is used seamlessly for the *transitive closure* construct. This is shown in the following piece of HLN code.

```
path UML!Class.directSuperclasses
       select generalization
       select general;
transitive closure allSuperclasses of UML!Class.directSuperclasses;
```

Another useful example of this technique involves defining the "inverse transitive closure" of a relationship, which can be computed combining the *opposite* and *transitive closure* constructs.

5 Related Work

Two main approaches to model query or navigation can be found in model transformation languages: patterns and navigation languages. Graph patterns are typically used in graph transformation languages, such as Viatra [4] or GReAT [3]. Objects patterns are available in QVT Relational [12] and in Tefkat [10]. OCL-like navigation languages are the primary navigation mechanism provided by rule transformation languages such as ATL [8], QVT Operational [12], and ETL [9].

Even though in this paper we have focused on OCL, our approach is applicable to complement other query languages. For instance, pattern languages use object properties to constraint query results. Such properties can be helpers defined by an HLN library, since the VM makes the integration seamless.

Regarding the performance of OCL, in [6] the need for developing benchmarks to compare different OCL engines is mentioned. The authors have developed several benchmarks but they are intended to compare features of OCL engines, rather than performance. In [11] the authors present several algorithms to optimize the compilation of OCL expressions. They argue that its optimizing OCL compiler for the VMTS tool can improve the performance of a validation process by 10-12%.

Finally, domain-specific query languages have been proposed as a means to enhance the query mechanism (strategies) of the Stratego program transformation tool. In particular, the implementation of an XPath-like language is discussed [13]. It behaves like a macro-system, generating Stratego code. In our case, we are able to generate efficient VM code.

6 Conclusions

In the paper, we reported on our experiments with an approach to model navigation based on a high-level navigation language providing declarative constructs. We introduced the HLN language, and we compared HLN against writing navigation expressions in OCL. The initial benchmarks we have carried out have shown performance improvements ranging from 20% to 800% with respect to a normal ATL implementation.

The contribution of this work is two-fold, on the one hand we have shown that raising the level of abstraction of a navigation language has several advantages: (1) it allows the compiler to easily optimize, which yields improved performance, (2) quality attributes such as readability and maintainability are also improved, and (3) model navigation best-practices are encoded in language constructs. On the other hand, the HLN implementation is a contribution itself. Its current implementation can be used as a complement to any language built on top of the ATL VM, and it is also useful for tool implementors to compare the performance of their navigation languages[2].

A possible extension of this work includes adding new constructs and improving performance of the current ones, comparing to other transformation languages (e.g., with an optimizing compiler), and creating a static type checker. It would also be useful to investigate how to improve interoperability of HLN with other languages (e.g., via parameters) while still keeping the language simple.

[2] The implementation of HLN, and several benchmarks can be downloaded from http://www.modelum.es/projects/hln [1].

Acknowledgments

This work has been supported by Fundación Seneca (Murcia, Spain), grant 08797/PI/08.

References

1. Benchmarks for HLN, http://www.modelum.es/projects/hln/
2. Specification of the ATL Virtual Machine, http://www.eclipse.org/m2m/atl/doc/
3. Agrawal, A.: Graph Rewriting and Transformation (GReAT): A Solution for The Model Integrated Computing (MIC) Bottleneck. In: ASE, pp. 364–368 (2003)
4. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: SAC 2006: Proceedings of the, ACM symposium on Applied computing, pp. 1280–1287. ACM, New York (2006)
5. Cuadrado, J.S., Jouault, F., Garcia-Molina, J., Bèzivin, J.: Optimization patterns for OCL-based model transformations. In: Proceedings of the 8th OCL Workshop (2008)
6. Gogolla, M., Kuhlmann, M., Buttner, F.: A benchmark for OCL engine accuracy, determinateness, and efficiency. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 446–459. Springer, Heidelberg (2008)
7. Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE 2006: Proceedings of the 5th International Conference on Generative programming and Component Engineering, pp. 249–254. ACM, New York (2006)
8. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
9. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
10. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
11. Mezei, G., Levendovszky, T., Charaf, H.: An optimizing OCL compiler for metamodeling and model transformation environments. In: Software Engineering Techniques: Design for Quality, pp. 61–71. Springer, Heidelberg (2007)
12. OMG. Final adopted specification for MOF 2.0 Query/View/Transformation (2005), www.omg.org/docs/ptc/05-11-01.pdf
13. van Wijngaarden, J.: Code Generation from a Domain Specific Language. Designing and Implementing Complex Program Transformations. Master's thesis, Utrecht University, Utrecht, The Netherlands, INF/SCR-03-29 (July 2003)
14. Vaziri, M., Jackson, D.: Some Shortcomings of OCL, the Object Constraint Language of UML. In: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 2000), Washington, USA, p. 555. IEEE Computer Society, Los Alamitos (2000)
15. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Precise Modeling With UML. Addison Wesley, Reading (1998)

Using Metrics for Assessing the Quality of ASF+SDF Model Transformations*

Marcel F. van Amstel[1], Christian F.J. Lange[2], and Mark G.J. van den Brand[1]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
{M.F.v.Amstel,M.G.J.v.d.Brand}@tue.nl
[2] Federal Office for Information Technology, Cologne, Germany
mail@christian-lange.com

Abstract. Model transformations are an essential part of Model Driven Engineering and are in many ways similar to traditional software artifacts. Therefore it is necessary to define and evaluate the quality of model transformations. We propose a set of six quality attributes to evaluate the quality of model transformations. We define 27 metrics for ASF+SDF model transformations to predict the quality attributes we propose. Metrics data has been collected from six heterogeneous model transformations automatically. The quality of the same transformations has been evaluated manually by several ASF+SDF experts. We assess whether the automatically collected metrics are appropriate predictors for the quality attributes by correlating the metrics data with the expert data. Based on the measurement results, we identify a set of predicting metrics for each of the quality attributes for model transformations.

1 Introduction

Model Driven Engineering [1] (MDE) is a software engineering discipline in which models play a central role throughout the entire development process. MDE combines domain-specific modeling languages for modeling software systems and model transformations for synthesizing them. Model transformations are in many ways similar to traditional artifacts, i.e., they have to be used by multiple developers, have to be changed according to changing requirements and should preferably be reused. Therefore, it is necessary to define and assess their quality. Quality attributes such as modifiability, understandability and reusability need to be understood and defined in the context of MDE, in particular for model transformations. For most other types of software artifacts, e.g. source code and models, there already exist approaches for measuring their quality. The goal of our research is to make the quality of model transformations measurable. In this paper, we focus on model transformations created using the ASF+SDF [2] term

* This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute with Vanderlande Industries as the industrial partner. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

rewriting system which is actively applied in MDE projects [3,4,5]. However, we expect that our approach can be generalized and applied to other model transformations formalisms such as ATL [6], QVT [7] and openArchitectureWare [8].

We propose the following six quality attributes for measuring the quality of model transformations: *understandability*, *modifiability*, *reusability*, *modularity*, *completeness*, and *consistency*. Most of these quality attributes have already been defined earlier for software artifacts in general [9]. In [10], we explain why they are relevant for model transformations in particular. We also define 27 metrics for ASF+SDF transformations as predictors for these quality attributes. Some of these metrics are specific for ASF+SDF only, but for most metrics a conceptually equivalent metric can be defined in other model transformation formalisms as well. To assess whether the metrics are valid predictors for the quality attributes, an empirical analysis has been conducted. Metrics have been collected from a total of six transformations by a tool we created. The same cases have also been manually assessed by ASF+SDF experts. We correlate the metrics data with the expert data to explore the relations between the metrics and the quality attributes. In this way we can assess whether the automatically collected metrics are appropriate predictors for the quality attributes.

The remainder of this paper is structured as follows. In Section 2, the metrics we define to predict the quality attributes are described. Section 3 describes the results of our empirical study. Section 4 describes related work. Conclusions and directions for further research are given in Section 5. Note that we will not describe ASF+SDF here. Instead, the reader is referred to [2]. For an extended version of this work the reader is referred to [10].

2 Metrics

This section describes the metrics we defined and measure with a tool we implemented. The metrics described here are specific for ASF+SDF. However, for most of them a conceptually equivalent metric can be defined for other model transformation formalisms as well. All metrics are listed in Table 2.

2.1 Transformation Function Metrics

A measure for the size of a model transformation is the number of transformation functions it encompasses. A transformation function in ASF+SDF consists of one or more signatures and one or more equations. The *number of transformation functions* is therefore defined as the number of signatures that are implemented by at least one equation. The size of individual transformation functions can be measured by the metrics *number of signatures per function* and *number of equations per function*. These metrics measure the number of variants of a transformation function. Equations may have conditions. We measure the size of an equation as the *number of conditions* it has. Conditions can also be included when measuring the size of a transformation function. This leads to the metric *number of equations and conditions per function*. In this case, the number of variants of a transformation function is measured along with their sizes.

Measurements for the complexity of a transformation function are the number of arguments it takes and the number of values it returns. In ASF+SDF, a transformation function can be overloaded by defining multiple signatures and equations for it. These signatures may have different arguments. We measure the average number of arguments of a transformation function. This metric is called *val-in*. In ASF+SDF, a transformation function can return only one value. Therefore it does not make sense to measure the number of return values of a transformation function, i.e., val-out. However, different signatures of an overloaded transformation function may return values of different types. Therefore, we measure the *number of distinct return types per function*.

Transformation functions generally depend on other transformation functions. To measure this dependency, we measure *fan-in* and *fan-out* of transformation functions. Fan-in of a transformation function f is the number of times f is invoked by another transformation function f'. Fan-out of a transformation function f is the number of times f invokes another transformation function f'.

In ASF+SDF, there are a few mechanisms to influence the flow of control of the transformation engine. These are, amongst others, conditions, default equations and traversal functions. Two types of conditions can be distinguished, viz. matching conditions and (in)equality conditions. We measure how often the different condition types are used, by measuring the *number of matching conditions per equation* and the *number of (in)equality conditions per equation*. The number of matching conditions is of particular interest. It is possible to write equations that express the same in different ways. One can either write relatively small equations with a relatively large number of matching conditions, or relatively large equations with relatively few matching conditions. Upon evaluation, default equations are always evaluated last. Transformation functions without a default equation may be incomplete and hence may not rewrite properly. Therefore, we measure the *number of default equations per function*. Traversal functions can also be used to change the way evaluation of a transformation function is performed. A traversal function visits every node of a tree once, whereas a standard transformation function is applied to one node only. In this way, traversal functions allow a collapse of the number of transformation functions corresponding to a syntax directed translation scheme. Therefore we distinguish *traversal functions* when measuring the number of transformation functions. In other transformation formalisms, different mechanisms are used to influence the transformation engine. For example, in ATL it is possible to use lazy matched rules. A standard matched rule is applied only once, whereas a lazy matched rule is applied as often as it is referred to [6].

2.2 Module Metrics

Most model transformation formalisms enable a modular definition of model transformations. This is also the case for ASF+SDF. The *number of modules* is a measure for the size of a model transformation. The size of an individual module can be measured in different ways. We introduce three metrics to measure the size of a module, viz. the *number of transformation functions per module*, the *number of signatures per module* and the *number of equations per module.* These metrics can be compared with the average values over all modules to assess the balance of a module with respect to the rest of the model transformation.

Dependencies between modules can be measured on a module level. A module m depends on another module m' if module m imports module m'. To measure this type of dependency between modules, we measure the *number of import declarations per module* and the *number of times a module is imported* by other modules. Dependencies between modules can also be measured on the level of transformation functions. Transformation functions may invoke transformation functions defined in other modules. To measure this type of dependency between modules, we measure *fan-in* and *fan-out* for modules. Fan-in of a module m is the number of times a transformation function defined in module m is invoked by a transformation function defined in another module m'. Fan-out of a module m is the number of times a transformation function defined in module m invokes a transformation function defined in another module m'.

2.3 Consistency Metrics

A transformation function in ASF+SDF consists of signatures and equations. Each signature is related to one or more equations. A signature may have no related equations. This can for instance occur when a transformation is still under development. To detect this inconsistency, we measure the *number of signatures without equations.* An equation that is not related to a signature will be detected by ASF+SDF itself. Therefore we do not measure this.

Variables are usually defined in a `hiddens` section. This means that they can only be used in the module they are defined in. Therefore, a variable needs to be redefined if it is to be used in other modules. This may lead to inconsistencies in variable naming, i.e., a variable name in one module can be related to a different type in another module, or vice versa. It may also cause (re)definition of variables that are not used in a module. To detect these inconsistencies, we measure the *number of (different) variable names per type*, the *number of (different) types per variable name* and the *number of unused variables.* A variable is unused in a module if there are no instances of it used in the module it is defined in.

3 Empirical Exploration of the Metrics

The quality attributes relevant for the evaluation of model transformations in practice are not directly measurable. Therefore, we are interested in the relation between metrics and quality attributes. The purpose of the case study described in this section is to explore this relation. In the case study, we used six model transformations specified in ASF+SDF. For each of these transformations we collected metrics data. To evaluate the quality attributes for each of the transformations directly, we used a questionnaire that was completed by four experts in ASF+SDF. In this section we describe the design of the case study and the statistical analysis and interpretation of the collected data.

3.1 Objects, Subjects, Task and Instrumentation

The experimental objects are six model transformations specified in ASF+SDF. These transformations are real-world transformations created by different developers in research projects. The transformations differ in size, style, structure and functionality. Table 1 summarizes the characteristics of the transformations.

Table 1. Characteristics of the analyzed model transformations

Transformation	LOC	# Functions	Purpose	Reference
ACP2UML	5694	173	Transform process algebra models into UML	[3]
SL2XMI	1851	70	Transform surface language into activities	[4]
SLCheck	1430	58	Surface language wellformedness checker	[4]
ASF2C	7096	396	Generate C code from ASF specifications	[11]
UML2DOT	1553	28	Transform UML activities into the DOT language	–
REPLEO	4058	47	Syntax-safe template engine	[5]

The subjects in the study were four experienced users of ASF+SDF. All subjects are researchers who have developed several ASF+SDF transformations. None of the authors participated as subject in this study. Prior to their task, the subjects were not informed about the particular purpose of the study. Their task was to answer a questionnaire consisting of 23 questions. The questionnaire contained at least three similar, but different questions for each of the quality attributes. In each question, the subjects had to indicate their evaluation of one of the quality attributes on a five-point Likert scale (1 indicating a very low value and 5 indicating a very high value). The questionnaire can be found in [10]. For each of the six transformations, the subjects used the same questionnaire. Five transformations were evaluated by three subjects, the transformation "ASF2C" was evaluated by all four subjects. During the evaluation, the subjects had the transformation opened in the ASF+SDF Meta-Environment [12] on their own computer. There was no time-bound for the evaluation task. In addition to the quantitative evaluation of each of the transformations, a semi-structured interview was conducted after the questionnaire task to obtain qualitative statements.

The metrics were collected using the metrics collection tool we implemented. We collected the data without taking library modules into account since library modules can severely affect the analysis results.

3.2 Relating Metrics to Quality Attributes

To establish the relation between metrics and quality attributes we analyze the correlation between them. The data acquired from the questionnaire is ordinal. Therefore, we use a non-parametric rank correlation test [13]. Since the data set is small and we expect a number of tied ranks, we use Kendall's τ_b rank correlation test [14]. This test returns two values, viz. significance and correlation coefficient. The significance indicates the probability that there is no correlation between metric and quality attribute even though one is reported, i.e., the probability for a coincidence. Since we are performing an exploratory study and not an in-depth study, we accept a significance level of 0,10. The correlation coefficient indicates

Table 2. Kendall's τ_b correlations

		Understandability		Modularity		Modifiability		Reusability		Completeness		Consistency	
#	Metric	CC	Sig.	CC	Sig.	CC	Sig.	CC	Sig.	CC	Sig.	CC	Sig.
1	# Transformation functions	-,550	,002	,439	,017	-,303	,092	-,242	,183	,053	,772	-,190	,307
2	# Traversal Functions	-,356	,048	,159	,385	-,432	,016	-,216	,235	,053	,772	-,136	,465
3	# Modules	-,801	,000	,713	,000	-,553	,003	-,256	,166	-,164	,379	-,483	,011
4	# Signatures without equations	-,481	,009	,631	,001	-,146	,424	-,135	,466	-,150	,420	-,476	,012
5	# Functions per module	-,278	,123	,279	,128	-,032	,858	-,111	,541	,106	,563	-,054	,770
6	# Traversal functions per module	-,006	,971	-,120	,515	-,058	,747	-,163	,369	,093	,613	,149	,422
7	# Signatures per module	-,123	,495	,120	,515	,123	,496	-,059	,746	,106	,563	,027	,884
8	# Equations per module	,667	,000	-,638	,001	,381	,035	,177	,331	,212	,247	,570	,002
9	# Signatures per function	,032	,858	,040	,828	,019	,914	-,033	,857	-,172	,347	-,203	,274
10	# Equations per function	-,265	,141	,279	,128	-,213	,237	,033	,857	-,146	,426	-,285	,125
11	# Default equations per function	,084	,641	-,080	,664	-,161	,370	,098	,590	-,053	,772	-,041	,827
12	# Eqs. and conditions per function	,175	,332	-,093	,612	,252	,162	,229	,208	-,040	,828	-,068	,715
13	Function fan-in	-,136	,451	,106	,562	-,058	,747	,111	,541	-,040	,828	-,176	,343
14	Function fan-out	,019	,914	,066	,717	,097	,591	,177	,331	-,040	,828	-,149	,422
15	Module fan-in	-,601	,001	,690	,000	-,307	,104	-,195	,306	-,073	,703	-,374	,054
16	Module fan-out	,188	,298	-,199	,277	,432	,016	,059	,746	,106	,563	,271	,144
17	# Conditions per equation	,550	,002	-,518	,005	,432	,016	,190	,297	,238	,193	,488	,009
18	# Matching conditions per equation	,693	,000	-,571	,002	,587	,001	,268	,140	,172	,347	,407	,029
19	# Equality conditions per equation	-,278	,123	,372	,043	-,097	,591	-,072	,692	,026	,885	-,068	,715
20	# Distinct return types per function	-,537	,003	,465	,011	-,355	,049	-,150	,408	-,093	,613	-,515	,006
21	# Import declarations per module	-,162	,370	,319	,082	,148	,410	-,059	,746	-,172	,347	-,258	,165
22	# Times a module is imported	-,655	,000	,658	,000	-,393	,032	-,175	,343	-,232	,213	-,566	,003
23	# Variables per type	-,758	,000	,678	,000	-,432	,016	-,268	,140	-,053	,772	-,407	,029
24	# Distinct variables per type	-,758	,000	,678	,000	-,432	,016	-,268	,140	-,053	,772	-,407	,029
25	# Types per variable	-,658	,000	,631	,001	-,378	,045	-,152	,426	-,190	,321	-,584	,003
26	# Unused variables per module	-,291	,106	,332	,070	-,071	,694	-,124	,494	,053	,772	-,054	,770
27	Average val-in	-,123	,495	,080	,664	-,136	,452	-,098	,590	-,172	,347	-,231	,215

CC: Correlation coefficient
Sig.: Two-tailed significance

the strength and direction of the correlation. A positive correlation coefficient means that there is a positive relation between metric and quality attribute and a negative correlation coefficient implies a negative relation. Note that correlation does not indicate a causal relation between metric and quality attribute. Table 2 contains the correlations we acquired. The significant correlations are marked.

No metric correlates significantly with reusability. The reason for this is that the experts cannot evaluate reusability properly because they do not see what they can reuse parts of the transformations for. Also, no metric correlates significantly with completeness. The reason for this is that the experts could not evaluate completeness properly because they did not have the specification of the analyzed transformations. Moreover, the time needed to get acquainted with the source and target language of the transformations is large.

The metrics that indicate the size of a transformation, i.e., *number of (traversal) functions* and *number of modules* correlate negatively with both understandability and modifiability. This indicates that larger model transformations are harder to understand and to modify. The same size metrics correlate positively with modularity. In larger transformations the need for splitting functionality over modules becomes higher. Therefore, a larger transformation often implies more modules, and therefore a more modular transformation. However, a high number of modules alone is not enough for a model transformation to be modular. Functionality should be well-spread over these modules.

The metric *number of modules* correlates negatively with consistency. More modules often implies a more complex transformation and more interfaces between modules. This may lead to inconsistencies. Also, when multiple developers work on a transformation it is likely that they work on separate modules. Since every developer has his own style, this may lead to inconsistencies.

The *number of (matching) conditions per equation* is positively correlated with understandability and modifiability. When writing equations, a tradeoff has to be made between writing a complex equation with little matching conditions or writing a simple equation with more matching conditions. The correlation indicates that simple equations with more matching conditions are preferred.

The *number of equations per module* correlates negatively with modularity. Modularity means that functionality should be spread over modules. This usually leads to smaller modules, i.e., modules with fewer equations.

In transformations consisting of multiple modules, modules depend on each other. This is expressed by module fan-in, module fan-out, the number of times a module is imported, and the number of import declarations. Therefore, *module fan-in* and *number of times a module is imported* correlate significantly in a positive way with modularity. These two metrics correlate negatively with modifiability. When a module on which other modules depend needs modifications, attention should be paid that these dependencies remain correct.

The *number of distinct return types per function* correlates negatively with modifiability and consistency. A function with multiple return types has multiple equations. This has two disadvantages with respect to modifying a transformation. First, if only one, or a few equations need modifications, attention should be paid that the correct equation is modified. Second, more equations imply more modifications. The correlation with consistency is to be expected because the return types are not consistent with each other.

The *number of types per variable* also correlates with consistency negatively. This is to be expected, because a variable that is of a different type in different modules is inconsistently defined. Related to this is the negative correlation between the number of (distinct) variables per type on consistency. Redefinition of variables may lead to inconsistent naming. In fact, the metric *number of (distinct) variables per type* measures this directly.

3.3 Threats to Validity

Conducting empirical studies involve threats to validity. Here we discuss how we addressed potential threats to validity in the presented study.

An important issue that must be taken into account for empirical studies is the representativeness of the experimental design with respect to practice.

We selected experienced ASF+SDF experts as subjects in our study. Since our experience shows that model transformations are developed and maintained by experts in practice, we exclude the subject experience as a threat to the validity in our study. The transformations used as objects in our study are designed in and applied for practical purposes. Additionally, our sample of transformations is heterogeneous with respect to several characteristics. Hence, we do not consider the object selection as a threat to the validity of this study.

Our choice for the transformation formalism ASF+SDF could be discussed. Future replications of our study must prove whether the findings for ASF+SDF presented in this study will also hold for conceptually similar metrics for other transformation formalisms. In our study the objects conducted an evaluation task that is not representative for practical model engineering tasks, and therefore this is a potential threat to the validity. We addressed this threat in the design of our study by using at least three self-controlled questions for each quality attribute and we used the evaluations of four experts. The results were relatively consistent between the experts and between the self-controlled questions, respectively. Therefore we minimized this threat to validity.

The number of observations in this study is rather small. This is a potential threat to the validity. It is difficult to find a larger number of experts in ASF+SDF for participation in such a study. We accepted this threat to the validity, because the study is only a first exploration of transformation quality metrics.

4 Related Work

The authors of [15] discuss characteristics of MDE that should be taken into account when developing a quality framework for it. They also define a quality framework for MDE themselves. Like us, the authors recognize the need for evaluating the quality of model transformations. Therefore, they apply their framework to this matter. This results in a set of quality attributes and a suggested method for assessing them. Our approach complements theirs. The quality attributes we propose include the ones they propose for model transformations. In addition, we present metrics for assessing the quality attributes we defined.

In [16], a set of metrics is proposed to monitor iterative grammar development in SDF. The authors took metrics developed by others that are applicable to measure (E)BNF grammars and adapted them such that they can be used to measure SDF grammars. The difference with our work is that they focus on grammar development using SDF, whereas we focus on transformation development using both ASF and SDF.

In [17], metrics are defined for functional programming languages. Since ASF is a functional language, we were able to adapt some of the metrics they defined such that they can be used to measure the quality of model transformations.

We assess the relation between metrics and quality attributes empirically. Multiple experiments that use a similar approach are described in [18].

5 Conclusions and Future Work

5.1 Conclusions

We have addressed the necessity for a methodology to analyze the quality of model transformations. In this paper, we proposed six quality attributes to evaluate the quality of model transformations. We also defined a set of 27 metrics for predicting these quality attributes for model transformations created using ASF+SDF. These metrics can automatically be collected from ASF+SDF model transformation specifications by a tool we created.

For the evaluation of quality attributes of model transformations, it is necessary to be able to select appropriate metrics as indicators for the quality attributes. Our study is a first step into this direction and provides data that supports the selection of metrics for particular quality attributes. For most of the proposed quality attributes we found metrics that correlate with them. This can, amongst others, be used to indicate possible points for improvements in model transformations.

5.2 Future Work

The manual assessment of the quality of the model transformations was carried out by four ASF+SDF experts. In the future, we would like to have feedback from more experts and on more cases. In that way, the results will be more significant. Also, we would like to perform a more in-depth statistical analysis.

In this paper, we focused on model transformations created using ASF+SDF. We expect that our techniques can be generalized and applied to different model transformation formalisms as well. Our focus will be on ATL [6]. The quality attributes will be the same, but the metrics to predict the quality attributes will differ. However, we expect that most metrics will be conceptually similar.

Once we have identified quality problems in model transformations, we can propose a methodology for improving their quality. This methodology will probably consist of a set of guidelines which, if adhered to, lead to high-quality model transformations.

Acknowledgements. We want to thank Mark Ligtvoet for his help with implementing the tool and Jan Stoop for his help with the statistical analysis. We also want to thank the ASF+SDF experts for participating in this research.

References

1. Schmidt, D.C.: Model-driven engineering. Computer 39(2), 25–31 (2006)
2. van Deursen, A.: An overview of ASF+SDF. In: Language Prototyping: An Algebraic Specification Approach. AMAST Series in Computing, vol. 5, pp. 1–29. World Scientific Publishing, Singapore (1996)
3. van Amstel, M.F., van den Brand, M.G.J., Protić, Z., Verhoeff, T.: Transforming process algebra models into UML state machines: Bridging a semantic gap? In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 61–75. Springer, Heidelberg (2008)

4. Engelen, L.J.P., van den Brand, M.G.J.: Integrating textual and graphical modelling languages. In: Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (2009)
5. Arnoldus, B.J., Bijpost, J., van den Brand, M.G.J.: REPLEO: a syntax-safe template engine. In: Proceedings of the 6th international conference on Generative programming and component engineering, pp. 25–32. ACM, New York (2007)
6. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
7. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation specification. OMG Document formal/2008-04-03, OMG (2008)
8. Völter, M.: OpenArchitectureWare: a flexible open source platform for model-driven software development. In: Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference (2006)
9. Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., Macleod, G.J., Merrit, M.J.: Characteristics of Software Quality. North-Holland, Amsterdam (1978)
10. van Amstel, M.F., Lange, C.F.J., van den Brand, M.G.J.: Evaluating the quality of ASF+SDF model transformations. CS-report, Eindhoven University of Technology, Eindhoven, The Netherlands (2009)
11. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling rewrite systems: The ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (2002)
12. van den Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF meta-environment: A component-based language development environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
13. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous & Practical Approach, 2nd edn. PWS Publishing Co. (1996)
14. Field, A.: Discovering Statistics using SPSS, 2nd edn. Sage, Thousand Oaks (2005)
15. Mohagheghi, P., Dehlen, V.: Developing a quality framework for model-driven engineering. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 275–286. Springer, Heidelberg (2008)
16. Alves, T., Visser, J.: Metrication of SDF grammars. Technical report, Departamento de Informática da Universidade do Minho, Braga, Portugal (2005)
17. Harrison, R.: Quantifying internal attributes of functional programs. Information and Software Technology 35(10), 554–560 (1993)
18. Lange, C.F.J.: Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML. Ph.D thesis, Eindhoven University of Technology, Eindhoven, The Netherlands (2007)

Achieving Rule Interoperability Using Chains of Model Transformations

Marcos Didonet Del Fabro[1], Patrick Albert[1], Jean Bézivin[2], and Frédéric Jouault[2]

[1] ILOG, an IBM Company
[2]AtlanMod Group (INRIA & EMN)
{marcos.ddf,albertpa}@fr.ibm.com,
{jean.bezivin,frederic.jouault}@inria.fr

Abstract. Model Driven Engineering (MDE) is rapidly maturing and is being deployed in several situations. We report here on an experiment conducted in the context of ILOG, a leader in the development of Business Rule Management Systems (BRMS). BRMSs aim at enabling business users automating their business policies. There is a growing number of BRMS supporting different languages, but also a lack of tools for bridging them. In this paper, we present an approach based on MDE techniques for bridging rule languages; the solution has been fully implemented and tested on different BRMS. The success of the experiment has led to the development and chaining of a significant number of model transformations – no less than twenty. At the same time, this deployment has shown new problems arising from the management of a high number of artifacts. We discuss the positive assessment of MDE in this field, but also the need to address the complexity generated.

1 Introduction

This paper describes an experiment to apply MDE techniques to the field of Business Rule Management Systems (BRMS). The experiment was conducted over one year within an industrial environment (at ILOG), with a set of open source MDE tools previously built at INRIA (the AMMA tool suite [18]).

The field of BRMS is characterized by a number of normative, open source, or proprietary systems and languages (ILOG JRules [8], JBoss Drools [11], Fair-Isaac Blaze Advisor [6], etc.), allowing the expression of various solutions to business problems at a high abstraction level, but with heterogeneous sets of capabilities and languages. Going from the initial problem (e.g., UML to Java translation) to a more general problem of DSL (Domain Specific Language) to DSL translation is an important step that we faced in this industrial case.

The question to answer in the project was thus about which help - if any - MDE could bring to provide interoperability between BRMSs. A first limited experiment had been conducted before to bridge normative business rule languages [1]. Though our project has a much greater size and complexity, the former has provided us with inspiration and reasonable hope for convergence.

Our goal was to take projects of the source BRMS (Drools) as input and to automatically produce ready-to-use projects in the target BRMSs (ILOG JRules) and particularly between languages in different abstraction levels. We developed two interoperability tools (i.e., bridges): a reasonably simple one from DRL (Drools Rule Language [11]) to IRL (ILOG Rule Language [8]), and a much more complex bridge that takes IRL rules as input and that generates Business Rules written in the ILOG "Business Action Language" (BAL) [8].

The experiment was much broader that typical one-to-one transformations, because we had a higher number of artifacts that should be produced (e.g., different kinds of project or control files). In particular, two major issues have risen. First, we had to do an inventory of all artifacts and to create all the operations/transformations to produce them. Second, we had to coordinate their execution in a coherent flow. We provide a detailed description of the signature of the operations that we created, and how they were composed to handle these issues.

To summarize, the major contributions of this paper are the following. We report on an experiment that included a complex MDE architecture encompassing a large number of input or output artifacts. We present in detail all the operations needed to transform a technical into a business language. We apply our solution to different projects, going from standard business rules benchmarks to real-life interoperability project.

This paper[1] is organized as follows. Section 2 briefly introduces the field of BRMS. Section 3 presents the interoperability solutions that have been developed. Section 4 describes the lessons learned. Section 5 concludes.

2 Business Rule Management Systems

A BRMS [20] is composed of several components supporting the definition, management, and execution of business rules. The rules are written in Domain-specific declarative languages as close as possible to the application domain and terminology. The technical details of the rules and application are hidden by a "business-level" front-end close to natural language.

This enables the policies owners to create and manage the rules by themselves with little to no support from software developers. Being largely independent from IT, the business users can react almost immediately to most policies changes by adding, removing, or changing rules. Well-formalized processes guarantee that new or modified rules are well managed.

2.1 Rules

The rules are written in domain specific languages, which may have different levels of abstraction. In this section we present an overview of business rules.

2.1.1 Production Rules

Production rules (or simply rules) are used to help business organizations automating their policies, providing properties such as reliability, consistency, and scalability. The rules are mostly made of *If-Then* statements involving predicates and actions about sets of business objects (as shown in Figure 1). Rules variables are bound to objects of a certain type; when the condition part recognizes that a set of objects satisfies its predicates, the action is triggered.

[1] An extended version of this paper is available as an INRIA Research Report, number 6747.

```
rule setInsuranceRatio {
   when  ?c : Client (age > 18 && age < 25);
  then  ?c.setInsuranceRatio (1.25);
}
```

Fig. 1. A simple production rule

2.1.2 Business Rules

The business rules approach is an attempt to bring the power of rules programming to business users willing to automate business policies.

The business rules are expressed in a language close to natural language that can be understood and managed by "business analysts", for instance, "***If** the age of the client is between 18 and 25, set the insurance ratio to 125%*". This Business-level language is compiled into a lower-level technical language, i.e., at execution level; the 'production rules' semantics remains unchanged. The "Business Layer" is composed of additional models supporting the definition of the rules, such as the business objects of the domain (e.g., *Client, age*) and the corresponding terminology (e.g., "the clieant", "the age of the client").

2.2 Drools and JRules

Drools is an open-source BRMS part of the JBoss foundation. Its rules language is called DRL, for Drools Rule Language [10]. In this paper, we focused on the technical rules of Drools.

JRules is the product developed and marketed by ILOG. It supports the two different levels of a full-fledged BRMS: the technical level targeted at software developers and the business action language targeted at business users. They are explained below.

Technical rules: the technical rules of JRules are written in IRL. The complete specification of IRL may be found at [9].

Business Action Language (BAL): hides implementation details of IRL, allowing business analysts to concentrate on the business logic (see an example in Figure 2).

```
definitions
  set 'client' to a client;
if
   the age of 'client' is between 18 and 25
then
   set the insurance ratio to 125%;
```

Fig. 2. Example of a rule in BAL

Business Object Model (BOM): defines the classes representing the objects of the application domain. A BOM is in fact a simplified ontology defining the application classes. The BOM is mapped to the execution objects that support the actual application data. In the simplest cases, the BOM can be automatically extracted from the classes definitions in Java, or C#.

Vocabulary (VOC): defines a text file with a controlled vocabulary that "closes" the rules syntax on a fixed set of allowed words and fragments. The business rules editor proposes only the valid terms while a business user creates or modifies a rule.

B2X and project files: the "Business To eXecution" model (B2X) describes how the logical elements represented in the BOM are actually mapped to physical data structures (XOM), i.e., it is used when there is not an implicit 1-to-1 mapping towards Java, C# or XML. Thus, the application developer might add new B2X constructs to specify the way the new BOM elements are implemented with the target programming language.

A JRules application has an additional project file - *.ruleproject* (RP) – which specifies the project parameters, the folder organization.

3 Rule Interoperability

Our goal is to transform a set of rules of a source BRMS, into an executable set of rules (and associated files) of a target BRMS.

The architecture follows the usual MDE pattern: **inject**, **transform, extract**, and relies on core MDE practices and technologies: Domain Specific Languages (DSLs) [18], metamodeling [18], model transformations [15] and projections across technical spaces (i.e., injections and extractions) [17]. We have used the AMMA tool suite: KM3 [14] for defining the metamodels, TCS [13] for defining the textual syntax, and ATL [12] for the model transformations. There are several other tool suites similar to AMMA (e.g., Epsilon [16] or oAW [19]). The closest one is oAW, which is also available on the Eclipse.org platform. AMMA and oAW share the same model-centric vision and only differ on some implementation choices.

In our scenario, we have two complementary objectives: translating a set of rules in DRL into IRL and translating IRL into BAL. We first produce the IRL rules from the DRL ones, and then we produce the BAL rules from IRL.

3.1 Model Management Operations

The bridge is composed of *twenty four* operations (as shown in Figure 3). The operations are identified by an initial letter depending on the category (e**X**ternal JRules operation, **I**njection, **E**xtraction and translation, XML-ification and refactoring **T**ransformations), plus an integer increment. The labels indicate the kinds of artifacts that are produced, for instance, a BOM model, or a BOM file.

An operation has the following signature:

$$<M_{OUT1} : MM_{OUT1}, \ldots , M_{OUTm} : MM_{OUTm}> = T[TS_{OUT}\text{-}TS_{IN}](<M_{IN1} : MM_{IN1}, \ldots , M_{INn} : MM_{INn}>).$$

T is the operation name; [TS_{OUT}-TS_{IN}] are markers with the output and input technical spaces. For instance, when injecting a textual file into a model, we mark the operation with [MDE-EBNF]. These markers are optional and are not used when specifying model transformations in the MDE technical space.

<M_{IN1} - M_{INn}> are the set of input models (n >= 1); the input models conform to the input metamodels <MM_{IN1} - MM_{INn}>; the input metamodels may be equal; M_{OUT1} - M_{OUTm} is the set of output models (m >= 1); the output models conform to the output metamodels <MM_{OUT1} – MM_{OUTn}>.

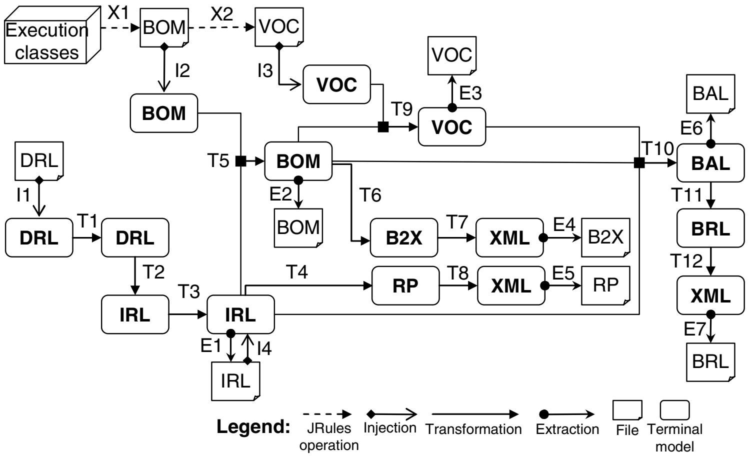


Fig. 3. DRL → IRL → BAL complete process

3.1.1 DRL to IRL Operations

This bridge produces an ILOG JRules project including rules written in the IRL language from a set of files written in DRL

$$drl_1 : DRL = I1[MDE\text{-}EBNF]\ (drl : DRL) \tag{1}$$

$$drl_2 : DRL = T1\ (drl_1 : DRL) \tag{2}$$

$$irl_1 : IRL = T2\ (drl_2 : DRL) \tag{3}$$

$$irl_2 : IRL = T3\ (irl_1 : IRL) \tag{4}$$

$$irl : IRL = E1[EBNF\text{-}MDE]\ (irl_2 : IRL) \tag{5}$$

The first operation (1) parses the textual file and it produces as output a model conforming to the DRL metamodel.

However, there are a few DRL expressions that are not natively supported by IRL: we first run an endogenous refactoring transformation (2) that takes a DRL model as input and that produces a refactored DRL model. Then, we translate the refactored DRL model into an IRL model (3). The IRL model is refactored as well (4).

This separation enables the specification of a relatively simple "DRL to IRL" transformation. Finally, the IRL models are extracted into the IRL files (5).

3.1.2 IRL to BAL Operations

This set of operations produces an ILOG JRules Project including rules written in the BAL syntax. As this bridge is far more complex than the previous one – it includes additional input and output models, such as the BOM, the VOC and the B2X. The bridge is a composition of twenty (20) operations.

$$bom : BOM = X1\ (java : JAVA) \tag{6}$$
$$voc : VOC = X2\ (bom : BOM) \tag{7}$$
$$bom_1 : BOM = I2[MDE\text{-}EBNF]\ (bom : BOM) \tag{8}$$
$$voc_1 : VOC = I3[MDE\text{-}EBNF]\ (voc : VOC) \tag{9}$$
$$irl_2 : IRL = I4[MDE\text{-}EBNF]\ (irl : IRL) \tag{10}$$

The first step in the bridge is the generation of the VOC and BOM - (6), (7). Though these operations could be defined using model transformations, we rather use the existing facilities provide by the JRules API. Then, the generated files, plus the input IRL files are injected into models - (8), (9), (10). Note that (10) is optional when the whole DRL→IRL→BAL bridge is executed, avoiding an extra injection.

$$bom_1 : BOM = T5\ (bom1 : BOM, irl_2 : IRL) \tag{11}$$
$$voc_1 : VOC= T9\ (bom_1 : BOM, voc_1 : VOC) \tag{12}$$
$$bom : BOM = E2[EBNF\text{-}MDE]\ (bom_1 : BOM) \tag{13}$$
$$voc : VOC = E3[EBNF\text{-}MDE]\ (voc_1 : VOC) \tag{14}$$

However, the initial vocabulary is not always complete. For instance, BAL does not natively support *insert* or *retract* actions and *for* and *while* statements. We augment the vocabulary and BOM - (11), (12) - to support these primitives. The augmented models are extracted - (13), (14) - overriding the initial files.

$$b2x_1 : B2X= T6\ (bom_1 : BOM) \tag{15}$$
$$rp_1 : RP = T4\ (irl_2 : IRL) \tag{16}$$
$$b2x\_xml_1 : XML = T7\ (b2x_1 : B2X\ ) \tag{17}$$
$$xml_2 : XML = E4[XML\text{-}MDE]\ (b2x\_xml_1 : XML\ ) \tag{18}$$
$$rp\_xml_1 : XML = T8\ (rp_1 : RP\ ) \tag{19}$$
$$xml_1 : XML = E5[XML\text{-}MDE]\ (rp\_xml_1 : XML\ ) \tag{20}$$

A B2X mapping model is produced with all the new methods (15). The project files are produced from the input IRL (16). However, the project files and the B2X mappings do not have a concrete textual syntax, i.e., they are saved in a specific XML format. We produce an XML model (XML-ification), conforming to an XML metamodel - (17), (18). The XML model is extracted into and XML file - (19), (20).

$$bal_1 :BAL=T10(irl_2 : IRL, bom_1 :BOM, voc_1 :VOC) \tag{21}$$
$$bal : BAL = E6[EBNF\text{-}MDE]\ (bal_1 : BAL) \tag{22}$$
$$brl_1 : BRL = T11\ (bal_1 : BAL\ ) \tag{23}$$
$$brl\_xml_1 : XML = T12\ (brl_1 : BRL) \tag{24}$$
$$xml_3 : XML = E7[XML\text{-}MDE]\ (brl\_xml_1 : XML\ ) \tag{25}$$

The central operation of the bridge is the transformation of the IRL models into the BAL models (21). The transformation must search the corresponding expressions in the BOM and in the vocabulary. Once the verbalization is found and transformed into the correct expressions (e.g., arithmetical expressions). Though the development of this transformation has been quite challenging, its code is out of the scope of this paper. The BAL models are extracted into their textual format (22). However, the BAL rules are encapsulated into one more XML format, called BRL. Thus, the BAL rules are transformed into BRL (23), which is in turn XML-ified - (24), (25).

3.2 Chaining and Parameterization

The correct chaining of transformations is an important factor of success of the project, because the bridge must be easy to configure and to run, acting over several input and output models. An operation cannot be fired until all its parameters are loaded. Consequently, the dependency relations shown in Figure 3 must be respected.

We use the AM3 [3] tool to create scripts that execute chains of transformations. AM3 provides a set of Ant tasks integrated with the Eclipse environment.

Consider the DRL to IRL bridge. The operations that have been defined take a fixed number of models/files as input and they produce a fixed number of files/models as output. However, a typical BRMS has hundreds or thousands of rules. Thus, the operations must be executed several times, and the files must be created in the correct folders. We illustrate below a script (using pseudo-code) that performs the transformation of several DRL files.

```
procedure DRL2IRLBridge() {
  Registry r = new Registry();
  Transformation t1 = r.newTransformation("DRLRefactor");
  Transformation t2 = r.newTransformation("DRL2IRL");
  Transformation t3 = r.newTransformation("IRLRefactor");
  FileList fList = readfiles("/Input/");
  For each file in fList {
     Model drl = r.inject(aFile, "DRL"); //I1
     drl = t1.execute(drl);
     Model irl = t2.execute(drl);
     irl = t3.execute(irl);
     r.extract(irl, "/Output/"); //E1
  }
}
```

This script is implemented using a combination of native Ant and AM3 tasks. We illustrate below three AM3-specific tasks: *loadModel*, *atl* and *saveModel*. The following task reads the text file, and injects it into a model. The task uses and the TCS injector implemented at *DRL-parser.jar*.

```
<am3.loadModel modelHandler="EMF" name="in_DRL" metamodel="DRL"
               path="/Inputfolder/drl1.drl">
  <injector name="ebnf">
     <param name="name" value="DRL"/>
     <classpath>
       <pathelement location="/DRL/Syntax/DRL-parser.jar"/>
     </classpath>
   </injector>
</am3.loadModel>
```

The task below executes the DRL2IRL transformation. We define the transformation path, the input and output models. The *name* attribute must be the same as the one declared in the transformation header. The *model* attribute uses the unique names that are previously affected to the models.

```
<am3.atl path="/DRL2IRL/DRL2IRL.atl">
  <inModel name="DRL" model="DRL"/>
  <inModel name="IN" model="in_DRL"/>
  <inModel name="IRL" model="IRL"/>
  <outModel name="OUT" model="out_IRL" metamodel="IRL"/>
</am3.atl>
```

Finally, the *saveModel* task extracts the transformed model (*out_IRL)* into the specified path. It uses the IRL TCS extractor in order to produce the IRL file.

```
<am3.saveModel model="out_IRL" path="/Output/out.irl">
  <extractor name="ebnf">
    <param name="format" value="IRL.tcs"/>
  </extractor>
</am3.saveModel>
```

4 Lessons Learned

This experiment has shown that MDE tools reached a reasonable level of maturity allowing their use in the context of industrial projects. The major advantage is the possibility to concentrate on the problem specification and to apply a declarative and modular approach using a small set of principles and tools.

The bridges have been first tested on the two standard business rules benchmarks: *manners* and *waltz* [2] and on an "insurance claim" demonstration [7]. We have run the *manners* benchmark with settings for 16, 32, 64, 128, 256 and 512 objects. Then, the bridges were applied to the migration of more than 100 Drools rules used by a banking application. The rules are executed over the Java objects provided with the examples. In all cases – DRL, IRL and BAL - the execution of the rules produced the same results. The execution was empirically validated. The IRL2BAL bridge has been used to translated 1500+ rules.

In table 1, we summarize the major difficulties found and how we solved them.

KM3 enables the definition of the metamodels with no particular limitations. On the syntax side, thought we could eventually reach our objective, we found more difficulties with TCS, because of its context-free approach. Such a limitation has introduced an unwanted level of complexity in our metamodels. We thus produced a BAL metamodel covering a large subset of the language. We could have used some existing code generation tool to produce the final BAL, because they have fairly good capabilities. However, using a bidirectional specification enabled the reutilization of the generated code into other applications.

Table 1. Summary of problems/solutions

Problem	**Solution**
BAL is context-sensitive.	Specification of the context-free part. This is enough for extraction and injection of a large subset.
Composed operators: « is not », « is not null », etc.	One element per operator : *IsNotOpt*, *IsNotNullOpt*, etc. Main drawback: increases the metamodel's size.
Languages with different abstraction levels.	Separation in two bridges. Other technical-language bridges can be plugged relatively easy.
Incompatible expressiveness of DRL and IRL	Creation of endogenous refactoring transformations. It simplifies the exogenous transformations.
Incomplete BOM and Vocabulary (IRL is more expressive than BAL).	Creation of augmentation transformations: inclusion of new logical methods in the BOM, plus a mapping to the execution layer.
Different input and output formats.	Definition of injections, extractions and XML-ification transformations.
Complex execution flow	Definition of automation scripts (Ant + AM3).
Several input/output files and folders.	Parameterization using property files. Theses files can be easily changed for different projects.
Performance : several input and output models.	Caching of operations (*helpers*) and saving the models only on demand.

We have applied different kinds of operations: injections, extractions, and transformations (XML-ifications, translation, augmentation and refactoring). The classification of transformation kinds enables the factorization of knowledge, in order to further study each kind of transformation separately.

Current solutions still need to improve their modularization capabilities. For instance, we would like to have transformations and libraries separated by packages, and with easy ways to navigate through the transformation code. We chose to modularize the transformation code using separate library of helpers.

We believe that it is rather rare to have a project that executes a single transformation. Thus, the utilization of a script language coupled with property files has proved very important in the usability of the bridges. The use of AM3 for writing the operation chaining (based on apache Ant tasks) as a basis was very useful because it is a well-know tool that can be learned relatively fast. However, an integrated repository, using graphical interfaces integrated within Eclipse would help a lot. As a side consideration, we can observe here the need to get tools to handle complex networks of transformations. This is a field where MDE has still to provide new solutions.

In addition, the scripts and the property files were not models, violating the base principle "everything is a model" of a complete MDE approach. We think the script language can be integrated into a megamodeling platform [5].

5 Conclusions

In this paper, we have presented an experiment to achieve interoperability between industrial BRMS. The utilization of MDE techniques enabled to successfully develop two bridges amongst rule languages with different degrees of expressiveness.

Our approach has been validated by executing the bridge on a set of well-known benchmarks for business rules, on a demonstrative example and on industrial applications. The bridges produced the expected results.

To the best of our knowledge, this is the first approach implementing a solution for transformations from a technical rule language into a business rule language with such a large number of transformations. The transformations defined could be implemented using different transformation languages. The discovery, development and chaining of complex transformations has been a challenging task. This project and particularly Figure 3, has been used by Don Batory [4], to illustrate the need for regular and complete frameworks for Model Driven Engineering.

The presence of several transformations, models and metamodels showed that megamodeling is a crucial issue when dealing with large projects. The scripting language is a first step that helped a lot, together with its parameterization. However, we think much more work can still be done on that area, especially in the specification of generic megamodeling platforms.

There are several possibilities for future work, such as the creation of similar bridges amongst different rule languages. The parsing can be internationalized into different languages, which is a common requirement of industry. Finally, we plan to study how to parse context-aware grammars, or even natural language.

The main conclusion of this work is that MDE has reached a level of maturity that allows using it to revisit traditional solutions to complex real-life problems and not only to toy examples. However, care should be taken to control the inherent complexity of the solution. New ways to manage important numbers of related modeling artifacts are among the most urgent needs to prepare model driven engineering for moving to full-scale industrial deployment.

Acknowledgments. This work has been partially supported by ANR IdM++ project.

References

1. Abouzahra, A., Barbero, M.: Implementing two business rule languages: PRR and IRL, `http://www.eclipse.org/m2m/atl/usecases/PRR2IRL/.03/07`
2. Academic Benchmark performances (27-12-2007), `http://blogs.ilog.com/brms/2007/10/22/academic-benchmark-performance/`
3. Allilaire, F., Bézivin, J., Brunelière, H., Jouault, F.: Global Model Management. In: Proc. of eTX Workshop at the ECOOP 2006, Nantes, France (2006)
4. Batory, D., Azanza, M., Saraiva, J.: The Objects and Arrows of Computational Design. In: Keynote talk at MODELS 2008 (2008)

5. Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: Proc. of OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop (2004)
6. Blaze Advisor (April 2008), `http://www.fairisaac.com/fic/en/product-service/product-index/blaze-advisor/`
7. Drools Examples (31-03-2008), `http://download.jboss.org/drools/release/4.0.4.17825.GA/drools-4.0.4-examples.zip`
8. ILOG JRules (October 2008), `http://www.ilog.com/products/jrules/index.cfm`
9. ILOG Rule Languages (November 2008), `http://www.ilog.com/products/jrules/documentation/jrules67`
10. JBoss Drools User Guide (15-01-2008), `http://downloads.jboss.com/drools/docs/4.0.4.17825.GA/html_single/index.html`
11. JBoss Drools (September 2008), `http://www.jboss.org/drools/`
12. Jouault, F., Allilaire, A., Bézivin, J., Kurtev, I.: ATL: a Model Transformation Tool. Science of Computer Programming 72(3), 31–39 (2008)
13. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proc. of GPCE 2006, Portland, Oregon, USA, pp. 249–254 (2006)
14. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
15. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
16. Kolovos, D., Paige, R., Polack, F.: A Framework for Composing Modular and Interoperable Model Management Tasks. In: Proc. of Workshop on Model Driven Tool and Process Integration (MDTPI), EC-MDA 2008, Berlin, Germany (2008)
17. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. In: Meersman, R., Tari, Z., et al. (eds.) CoopIS 2002, DOA 2002, and ODBASE 2002. LNCS, vol. 2519, Springer, Heidelberg (2002)
18. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL Frameworks. In: Proc. of Companion of OOPSLA 2006, Portland, OR, USA, October 22-26, pp. 602–616 (2006)
19. OpenArchitectureWare (November 2008), `http://www.openarchitectureware.com/`
20. Owen, J.: Business rules management systems. Extract business rules from applications, and business analysts can make changes without IT breaking a sweat. Infoworld (25-06-2004)

Bidirectional Transformations: A Cross-Discipline Perspective
GRACE Meeting Notes, State of the Art, and Outlook

Krzysztof Czarnecki[1],
J. Nathan Foster[2],
Zhenjiang Hu[3],
Ralf Lämmel[4],
Andy Schürr[5],
and James F. Terwilliger[6]

[1] University of Waterloo, Canada
[2] University of Pennsylvania, USA
[3] National Institute of Informatics, Japan
[4] Universität Koblenz-Landau, Germany
[5] Technische Universität Darmstadt, Germany
[6] Microsoft Research, USA

Abstract. The GRACE International Meeting on Bidirectional Transformations was held in December 2008 near Tokyo, Japan. The meeting brought together researchers and practitioners from a variety of subdisciplines of computer science to share research efforts and help create a new community. In this report, we survey the state of the art and summarize the technical presentations delivered at the meeting. We also describe some insights gathered from our discussions and introduce a new effort to establish a benchmark for bidirectional transformations.

1 Introduction

Bidirectional transformations (*bx*) are a mechanism for maintaining the consistency of two (or more) related sources of information. Researchers from many different areas including software engineering, programming languages, databases, and document engineering are actively investigating the use of *bx* to solve a diverse set of problems, for example:

– *Model-Driven Software Development*: to compute and synchronize views of software models [5,93,97,113].
– *Graphical User Interfaces*: to maintain the consistency of a GUI and the underlying application model in the model-view-controller paradigm [79].
– *Visualization With Direct Manipulation*: to visualize abstract data and animate algorithms [100].
– *Relational Databases*: to construct updatable views [10,16,27,59].

– *Data Transformation, Integration, and Exchange*: to map data across paradigms, merge it from multiple sources, and exchange it between sources [44, 47, 49, 57, 68, 82, 86].
– *Data Synchronizers*: to bridge the gap between replicas in different formats [17, 41, 58].
– *Macro Systems*: to give feedback to the programmer (e.g., from a type checker or a debugger) in terms of the original program elements prior to macro expansion [23, 62].
– *Domain-Specific Languages (DSLs)*: to translate between run-time values of the object language (the DSL) and the corresponding values of the host language in embedded interpreters [13, 88].
– *Structure Editors*: to provide convenient interfaces for editing complicated data sources [54, 55, 75].
– *Serializers:* to mediate between external data (binary or sequential data representations on the wire or the file system) and structured objects in memory [32, 39].

Although researchers are actively working on bidirectional transformations in several communities, so far there has been very little cross-discipline interaction and cooperation. The purpose of the GRACE International Meeting on Bidirectional Transformations (GRACE-BX), held in December 2008 near Tokyo, was to bring together international elites, promising young researchers, and leading practitioners to share problems, discuss solutions, and open a dialogue towards understanding the common underpinnings of *bx* in all these areas.

This report summarizes the main results of the GRACE-BX meeting and records the technical presentations that were delivered there. Section 2 surveys the essential ideas and important papers in key disciplines of *bx* and describes the specific work presented at the meeting. Section 3 identifies some of the common themes and cross-links that emerged during the meeting. Section 4 describes the beginning of an effort to develop a benchmark suite of canonical bidirectional transformations for comparing the capabilities and characteristics of different *bx* approaches, technologies, scenarios and implementations thereof. Section 5 concludes this report.

All of the material presented at the meeting is available at the GRACE-BX website.[1]

2 Bidirectional Transformation Subcommunities

Bidirectional transformations are being used in a variety of disciplines including programming languages, database management systems, model-driven engineering, and graph query and transformation systems. The precise details of how *bx* work in each area vary greatly between disciplines, but there is general agreement that a *bx* between two sources of information A and B (e.g., a database

[1] `http://grace.gsdlab.org/`

source and view, two different software models or graph structures, or the input and output of a program) comprises a pair of unidirectional transformations: one from A to B and another from B back to A.[2] In many cases, the flow of data from A to B dominates the flow of data from B to A—i.e., A acts as a master for B. In these cases, A is called the input (source, master) and B is called the output (target, slave) of the *bx*. Consequently, the transformation from A to B is often called the forward transformation and the transformation from B to A is called the backward or reverse transformation.

In current practice, *bx* are usually implemented by programming (or specifying) compatible forward and backward transformations in a *unidirectional* transformation language. Recently, however, an exciting new approach has started to be used in which *bx* are implemented in *bidirectional* transformation languages—formalisms where every program (or specification) describes a forward and a backward transformation simultaneously. A major advantage of this approach is that the compatibility of the transformations can be guaranteed by construction.

In the rest of this section, we give a brief overview of uses of *bx* in each community and describe the presentations that were delivered at the GRACE-BX meeting. We start with the area of bidirectional and reversible programming languages, and continue with databases and data management, model-driven engineering, graph query and graph transformation systems. We conclude this section by describing presentations on novel applications of *bx* including coupled grammar transformations and a system for computing updatable views of configuration files.

2.1 Programming Languages

Recently, a number of researchers have investigated programming languages where programs can be run both forwards and backwards. Broadly speaking, these languages can be classified according to two features: the *semantic laws* obeyed by programs and the *mechanisms* by which programs are made bidirectional.

At the level of semantics, the key distinction is between bijectiveness and bidirectionality. In *bijective languages*, the forward transformation denoted by every program is an injective function, and the backward transformation is the corresponding inverse. By contrast, in *bidirectional languages*, the forward transformation can be an arbitrary function. Since the forward transformation may discard information in general, the backward transformation typically takes two arguments: an updated output as well as the original input. It weaves these two together, yielding a new input where the information contained in the output has been propagated and the discarded information from the input has been restored. Bidirectional programs are typically required to obey "round-tripping" laws, which ensure that information is maintained by the transformation in each direction.

Several mechanisms for equipping programs with a bidirectional semantics have been investigated. In *reversible* models of computation, each step of

[2] Generalizations to more than two sources of information have also been studied, but are beyond the scope of this report; see [61] for an example.

computation is invertible [116]. Although these models can only realize injective functions, it has been shown that this does not represent a serious restriction—arbitrary functions can be made injective by adding the information discarded by the forward transformation, often called a complement, to its output. This approach is often used in *bidirectionalization* of unidirectional transformations. In general, there are many ways to represent the discarded information [12,55,77], and each results in a different backward transformation. Recently, semantic approaches to bidirectionalization have also been explored [108].

Other languages do not require explicit reversibility at each step. For example, programs in the languages biXid and XSugar, consist of pairs of intertwined grammars [17,58], and the transformations are obtained by parsing according to the rules in one grammar and pretty printing according to the rules in the other. Another approach, used in many bidirectional languages, is to have primitives that denote two transformations and combining forms that preserve bidirectionality [15,16,42,43,84].

Presentations at the Meeting

Holger Bock Axelsen gave two presentations on reversible models of computation. In his first talk, he described a general class of reversible abstract machine architectures and instruction sets. The individual instructions and control logic are both reversible and are based on a notion of reversible updates [9]. His second presentation described a result in computability theory: structured programming using a reversible flowchart language is as expressive as unstructured programming [117]. The result uses the concept of *r-Turing completeness*: a model of computation is r-Turing complete if it is capable of simulating Turing machines *cleanly*—i.e., without generating any extraneous output data. Unlike classical approaches [12], reversible flowcharts are r-Turing complete.

Kazutaka Matsuda described recent work on bidirectionalization—i.e., the task of deriving suitably-related backward transformations from the program describing the forward transformations. He described a method that, given a program written in ordinary λ-calculus notation, produces a corresponding well-behaved backward transformation [76]. The method derives "complement" functions and also uses an injectivity analysis to produce a backward transformation capable of handling a broad class of updates. Using a tupling transformation, the technique can also be cleanly extended to situations where the forward transformation duplicates parts of the input [77].

Janis Voigtländer presented a different approach to bidirectionalization that works purely semantically [108]. He described a higher-order function that takes a (polymorphic) forward transformation as an argument and constructs an appropriate backward function as a result. No special language for describing the forward transformation is needed, and the function that calculates the backward transformation does not inspect the program used to describe the forward function. The work is inspired by relational parametricity [89] and uses free theorems [111] to prove several round-tripping laws.

Meng Wang described an application of bidirectionalization for building algebraic views [112]. In this context, a *view* is an abstraction of the actual implementation of an abstract data type that provides an interface that is convenient for programmers to use and robust to changes [110]. Implementers of views are typically required to come up with pairs of conversion functions that are each other's inverses, a condition that is difficult for programmers to check and maintain. Using a bidirectionalization transformation, these properties can be obtained automatically.

Nate Foster presented work on lenses [15,16,42] that addresses the problem of handling inessential data in bidirectional transformations [43]. The foundation of most bidirectional languages is rooted in "round-tripping" laws which require that information be preserved in both directions. In practice, however, these laws are too strong: realistic bidirectional transformations do not obey them "on the nose," but only modulo equivalence relations that capture the essential parts of the data being manipulated. He described a general framework of *quotient lenses* and gave a syntax and a type system for building and reasoning about well-behaved quotient lenses.

Yingfei Xiong described Beanbag, a language for describing synchronization policies for software models [114]. Whenever one part of a model is modified, the Beanbag system computes the updates that need to be applied to the other parts of the model (as well as to related models) to reach a consistent state. Beanbag provides declarative syntax for describing inter-relations between models and intra-relations within a single model.

2.2 Databases and Data Management

In database literature, the fundamental unit of data movement or transformation is the *query*, which leverages the structure of potentially large quantities of data to provide declarative syntax, clean semantics, and efficient execution. The specification of a transformation may occur at some higher level such as schema matching, and is then compiled or converted into queries.

Until recently, most database research into *bx* studied whether an existing transformation specified in a query language, e.g., SQL, Datalog, or XQuery, can be "reversed" in some meaningful way, i.e., create a new transformation from the target of the original transformation to its source, rather than starting with bidirectionality by design, e.g., [27,37]. A database transformation between models $\mathcal{M}$ and $\mathcal{M}'$ can be bidirectional in two ways: *operationally* (by transforming both read and write operations on $\mathcal{M}$ into equivalent operations on $\mathcal{M}'$) or *by instance* (by determining to what degree instances of $\mathcal{M}$ can remain unaltered when being transformed into an instance of $\mathcal{M}'$ and back again).

Instance-at-a-time transformation is also known as *data exchange* in database literature [95,44]. The goal in data exchange is to transform instances of source model $\mathcal{S}$ into instances of target model $\mathcal{T}$ in such a way that a set of constraints $\Sigma_{\mathcal{ST}}$ that associate instances of $\mathcal{S}$ and $\mathcal{T}$ are satisfied. The constraints may be specified in whatever query or logical language suits the models, such as tuple-generating dependencies of first-order logic [38] or second-order logic [37] for relations or tree-based pattern matching for XML [7]. For a mapping $(\mathcal{S}, \mathcal{T}, \Sigma_{\mathcal{ST}})$ to be bidirectional, the goal is to construct another mapping $(\mathcal{T}, \mathcal{S}, \Sigma_{\mathcal{TS}})$ such that composing the two is the identity mapping [37]. Such inverses may not always exist, or may only be partial [8].

A concrete case of instance-at-a-time transformation is the *cross-metamodel mapping*, where the source and target models of a transformation are in different metamodels. In database research, the most common cases involve translating between three metamodels in particular: objects (O), XML (X), and relations (R) [68]. The O-R case has well-established commercially-available tools and mature research [20,60,81,86]. The X-O and X-R cases are substantially more difficult, given the difference in the expressive power of the metamodels [69]. There are several approaches to translating XML document instances into objects and vice versa. The most common cases involve either constructing XML-like objects that support an XML-like interface [50,80] or compiling an XML schema into a canonical class representation that supports translation of instances in either direction [65,66,73,115]. A more recent research effort allows declarative mappings to be specified between classes and XML schemas [82,103].

The operational case is often called the *view update problem*: given a target model $\mathcal{T}$ specified as a set of views over a source model $\mathcal{S}$ by a set of queries $\mathcal{Q}$, determine if it is possible to intercept updates to $\mathcal{T}$ and instead update $\mathcal{S}$ such that re-running queries $\mathcal{Q}$ on $\mathcal{S}$ regenerates the updated instance of $\mathcal{T}$ exactly. The update to $\mathcal{S}$ must be unambiguous, i.e., there can be exactly one way to do it [27]. Research into the view update problem generally focuses on identifying semantic or syntactic constraints on $\mathcal{Q}$ that can determine if a view defined using $\mathcal{Q}$ is updatable, and if the exact method of update translation can be determined strictly by examining syntax.

One area of research around updatable views pertains to what invariants should be respected with respect to view updates. For instance, the *constant complement* property suggests that not only should base data be unambiguously updatable, but that data not referenced by the view should remain constant [10].

A recent development in bidirectional database transformations is the study of smaller algebraic transformations with known properties. For instance, the foundation of both relational lenses [16] and Both-as-View [78] is a collection of atomic query transformations that are known to be bidirectional, from which more complex transformations can be built.

In the proximity of *bx*, there is a data management scenario for co-evolution of database schemas and database instances [48]. This is important when schema-level transformations (e.g., schema refactorings) need to be coupled with instance-level transformations so that existing instances can be adapted for use with a new schema. Likewise, transformations of XML schemas and XML-document transformations may be coupled [67]. Also, coupling is not limited to schema and instance transformations. In addition, co-transformation of schema-dependent programs may be an issue [22,107]. These are all instances of the notion of coupled transformations [14,64,106,107].

Presentations at the Meeting

Jácome Cunha covered HaExcel, which is a tool that correlates spreadsheets and databases, and prevents update anomalies in spreadsheet tables [24]. HaExcel uses data mining techniques to discover functional dependencies in spreadsheet data. These functional dependencies can be exploited to derive a normalized relational database schema. Finally, HaExcel applies data calculation laws to the derived schema in order to reconstruct a sequence of refinement steps that connects the relational database schema back to the tabular spreadsheet.

James Terwilliger presented two bidirectional frameworks; the first was Microsoft Entity Framework (EF), which serves as an object-relational mapping tool and is publicly available as part of the .Net framework [20]. In EF, the developer specifies a declarative mapping between an extended entity-relationship model and a relational database. This specification is then compiled into a pair of views (one view for query transformation, one for update transformation) that translates model operations into equivalent database operations [81].

Finally, James presented Guava, a system that treats the user interface of a business application as an updatable view of that system's database. Guava encapsulates middleware operations like data pivoting, unpivoting, merging and partitioning, tuple augmentation with environment data, and role-based security into algebraic operators. Each operator translates queries, updates, and schema modifications on its input into equivalent expressions on its output [101,102].

2.3 Model-Driven Engineering

Model-driven engineering (MDE) promotes the use of formal or structured specifications, which are referred to as *models*, with the goal of automating the derivation of implementations from such specifications. The approach involves many kinds of related models, such as requirements, design, and test specifications, and developers have to maintain complex relationships among the models and code. Examples of such relationships are *refinement* of design models to code and the *conformance* of models to their respective metamodels. Model transformations are mechanisms for establishing—and re-establishing, in the presence of change—the relationships among models and code [26]. Bidirectional model transformations are of particular interest if the related artifacts can be edited independently [5,98]. Such edits are necessary if different stakeholders require viewing information in different specialized notations, possibly at different abstraction levels, or some of the related artifacts contain independent pieces of information that cannot be derived from other artifacts or both.

The Object Management Group (OMG) acknowledged the importance of *bx* to MDE by including a *bx* language in their Query View Transformation (QVT) standard. Interestingly, Perdita Stevens analyzed the language from the viewpoint of the round-tripping laws (Section 2.1), pointing out several weaknesses. Other languages and systems to bidirectional model transformations have been proposed; see [30,45,51,113,114] for some more recent discussions and contributions.

The model-transformation scenario for co-evolution of metamodels and models, where metamodel adaptations must be coupled with corresponding model transformations [109], is closely related to *bx*. Such co-evolution is an instance of the notion of coupled transformations [64,106].

Presentations at the Meeting

Davide Di Ruscio gave a presentation on co-evolution of metamodels and models. This work is based on the axiom that metamodels must be considered one of the basic concepts of MDE and, accordingly, they are expected to evolve during their life cycle. As a consequence, models conforming to changed metamodels have to be updated to preserve their well-formedness. The presented work deals with the coupled transformation of metamodels and models by using higher-order model transformations which take a difference model for the metamodel level as input and produce a model transformation able to co-evolve the involved models as output [21].

Antonio Vallecillo gave a presentation on correspondences in viewpoint modeling for complex software systems. Viewpoint modeling is a technique for specifying software systems in terms of a set of independent viewpoints and correspondences between them. Correspondences specify the relationships between the elements in different views, together with the constraints that guarantee the consistency among these elements. Correspondences are hard to specify due to a lack of adequate notations, mechanisms, and tools. Also, specifications become unmanageable when the number of elements in a system is large. The presentation described efforts that are focused on the development of a generic framework and a set of tools to represent correspondences [104], which are able to manage and maintain viewpoint synchronization in evolution scenarios, as reported in [36]. The approach is based on modeling correspondences both intensionally and extensionally and the use of model transformation techniques to connect these two specifications.

Andrzej Wasowski gave a short progress report on his efforts to develop a flexible editing model for feature diagrams, including reversible editing steps and editing of broken (inconsistent) models. The presented topics included semantics for the editing process and algorithms for validation and guidance during modeling, under inconsistency. The aim of this work is to (ultimately) build a 'very intelligent' and highly flexible editor for feature models.

Bernd Fischer gave a short presentation of his work in progress on model-based code generation. The transformations used to generate code are fundamentally unidirectional because the generated code contains significantly more details than the model from which it was generated. This makes it hard to modify the generated code without causing model and code to get out of sync. In round-trip engineering, the direction of code generation it to be complemented by an inverse transformation (which is known to be a hard problem). In contrast, the presentation proposed to employ aspect-oriented techniques to achieve the desired code modifications by controlled modifications of the transformation. The core insight is that the concepts from the generator's domain model can be used to systematically derive the required join points.

Krzysztof Czarnecki first presented ongoing work (with Michal Antkiewicz) on an infrastructure for mapping domain-specific languages (DSLs) to code [2,4,6,71]. The infrastructure supports extracting domain-specific models from code as code views and bidirectional update propagation with conflict resolution. The extraction and update propagation rely on declarative mapping definitions, which relate elements of the DSL to structural and behavioral code patterns. The mapping definitions can be executed bidirectionally thanks to predefined code queries and code update transformations approximating the semantics of the mappings. The second presentation outlined an ongoing effort (with Zinovy Diskin and Michal Antkiewicz) to recast the design space of heterogeneous synchronization [5] in terms of synchronizers with category-theoretic underpinnings.

Zinovy Diskin gave a presentation on the algebraic foundations of model management (work in progress). Model management scenarios are often described by informal diagrams: nodes denote models and arrows are model transformations (of different types). The goal of the work is to reveal *diagrammatic* algebraic foundations underlying such diagrams and thus make the notation precise without translating it into formula-based formalisms. Some general ideas can be found in [29], and a recent promising application is described in [31].

2.4 Bidirectional Graph Transformation

The graph transformation research community, with its roots in the 1970's, can be considered today to be a special subarea of the model-driven engineering community. If we replace the terms "model", "metamodel", and "model transformation" by the related terms "graph", "graph type/schema", and "graph transformation", then it becomes obvious that graph transformation languages and tools are essentially model transformation languages and tools with a precisely defined semantics. Roughly speaking, one can identify three different families of graph transformations that either use category theory, non-standard logics, or set theory as their foundation. For a survey of related activities, we refer the reader to the LNCS Proceedings of the International Conference on Graph Transformation ICGT, as well as the set of so-called graph grammar handbooks [33,34].

One can distinguish at least two different sorts of bidirectional graph transformation approaches: (1) reversible graph transformation languages, which rewrite a given input graph step by step into a new output graph, and (2) truly bidirectional graph transformation languages, which manipulate pairs of graphs linked together by means of so-called *correspondence links*. All graph transformation languages that support the so-called "double pushout" category-theoretic graph transformation approach can be classified as reversible transformation languages. Conditions on rule application guarantee that all rewriting steps can be undone by simply applying the involved rewrite rule with exchanged left- and right-hand side [35]. *Triple Graph Grammars* (TGGs) [92,94] as the descendants of pair grammars [87] are a special brand of coupled grammars. They belong to the class of bidirectional transformation languages [25]. Pairs of uni-directional forward and backward transformations can be derived automatically from a given TGG that defines a language of related pairs of graphs.

Presentations at the Meeting

Andy Schürr gave a tutorial-style introduction to TGGs. TGGs are a bidirectional model transformation formalism, where a single specification generates a language of related graph tuples (pairs of models) together with an intermediate correspondence graph (traceability link database). A single TGG specification is used as input for a compiler that generates corresponding consistency checking, traceability link creating, and forward/backward model transformation implementations. The TGG tutorial reviewed the history of TGGs and sketched their formal definition relying on the theory of the algebraic/category-theoretic branch of graph grammars. Finally, the meta-modeling tool MOFLON was presented. MOFLON's implementation of TGGs adopts the visual notation of QVT Relational, the OMG standard bidirectional model transformation language.

Soichiro Hidaka, Hiroyuki Kato, Shin-Cheng Mu, and Keisuke Nakano gave presentations on different aspects of a functional approach to bidirectional graph transformation. This work aims at the development of an algebraic framework for bidirectional model transformation by integrating the state-of-the-art technologies on bidirectional tree transformations and the algebraic graph querying language UnQL+ [52,53], which is an extension of the known UnQL [18]. The theoretical foundation of the work is related to the family of category-theoretic graph transformations called *algebraic graph transformations*. The resulting bidirectional graph transformation approach comes with a powerful automatic bidirectionalization method for the automatic derivation of a backward graph transformation from a given forward graph transformation. For this purpose, a bidirectional semantics for an existing graph algebra based on structural recursion called UnCAL is used, which has been well studied in the database community. Hence, this work belongs to the class of reversible as well as to the class of truly bidirectional graph transformation languages. Moreover, the algebraic framework supports the systematic development of efficient large-scale bidirectional model transformations in a compositional manner.

2.5 Further Applications

GRACE-BX covered a number of presentations that we feel are best collected in a list of "further applications". It goes without saying that these applications regularly interact with issues of programming languages for *bx*, data management, or model-driven engineering.

Keisuke Nakano described the Vu-X approach to website construction that is based on bidirectional transformations [85] (as opposed to unidirectional transformations that simply translate data from a database into web content). Hence, users can directly modify a generated website, and the modification is automatically reflected in the database—without the need to update the database directly. The Vu-X system is also implemented as a web server so that users can edit it in WYSIWYG style within their web browsers.

Hui Song talked about runtime management of systems at the level of an intuitive, high-level architecture model [56,96]. Management agents use the architecture model to monitor and control a running system. A key component for architecture-based runtime management is the synchronizer that propagates changes between the architecture model and the system state, and maintains the correspondence between them. The presented approach supports automated generation of such synchronizers based on bidirectional transformations.

David Lutterkort presented a configuration API for Linux systems called Augeas [75]. Augeas parses configuration files in their native formats and transforms them into a tree. Configuration changes are made by manipulating this tree and saving it back into native configuration files. The string-to-tree transformation is specified by lenses so that some details can be left out from the tree level, but they are still preserved when writing back changes. For instance, Augeas makes an effort to preserve comments and formatting in the textual configuration files.

Kathleen Fisher presented recent work on PADS, a system for processing ad hoc data sources (such as log files) [39]. The PADS compiler takes declarative descriptions of data formats as input and generates a variety of software artifacts including a parser, an in-memory representation for the data, and a pretty printer, among others. The presentation described a mechanism for generating a suite of useful data processing tools, including a semi-structured query engine, several format converters, a statistical analyzer, and data visualization routines directly from ad hoc data, without human intervention [40]. The key technical contribution is a multi-phase algorithm that automatically infers the structure of an ad hoc data source and produces a format specification in the PADS data description language. Programmers wishing to implement custom data analysis tools can use such descriptions to generate printing and parsing libraries for the data.

Ralf Lämmel talked about grammar transformations [63,70]—specifically on coupled grammar transformations and their applications in XML-data binding and concrete/abstract syntax mapping. Grammar transformations are expressed in terms (of sequences) of primitive combinators, which can be applied both to grammars and instances (such as parse trees or documents). In the simpler, better understood cases, these grammar transformations are information-preserving, and an inverse is defined for each possible combinator. In more general cases such as mapping a rich concrete syntax to a more abstract syntax, bidirectionality is more difficult to achieve, subject to future work.

Zhenjiang Hu gave a presentation on the use of automatic function inversion as a means to obtain divide-and-conquer parallel programs from sequential programs [83]. This approach allows programmers to use the often more intuitive, sequential encoding style, while, under certain conditions, efficient parallel programs in the form of list homomorphisms can be derived automatically. These parallel programs would be more difficult to develop by programmers in the first place. The heavy lifting of the approach for the extraction of a list homomorphism is the automatic derivation of weak right inverses from sequential programs. Experimental results show the practical efficiency of the automatic parallelization algorithm and demonstrate that the generated parallel programs achieve good speedups.

3 Synthesis: Key Concepts and Properties of BX

The meeting included two discussion plenary sessions: one on the terminology and key concepts used across the represented communities and another on properties of *bx*. We briefly summarize the main discussion points in this section.

Lack of common and well-established terminology. The participants generally agreed that each represented community has developed its own terminology and that there is little sharing of terms across disciplines. Moreover, central terms such as "transformation" and "view" are overloaded. For example, "transformation" is sometimes used to mean "transformation specification", "transformation implementation", or "transformation execution". Likewise, although "view" has a precise and well-established meaning in databases, it is used in a different way in the programming languages community [110], and its meaning in MDE is not clear.

Transformation vs. synchronization. A hotly-debated topic was the distinction between transformation and synchronization. All of the participants shared a common understanding of transformations as executable operations on structured artifacts (data, models, programs) that establish well-defined relationships between the inputs and outputs, but the definition of synchronization was less clear. After discussion, a key difference was identified: synchronization (re-)establishes relationships among (partial) *replicas*, i.e., semantically overlapping artifacts that exist in parallel. Thus, although synchronization can be viewed as a kind of transformation, such as transforming code to make it consistent with an updated design, not all transformations do synchronization. For example, the reversible edits presented by Wasowski (see Section 2.3) are *bx* but not synchronizers since they relate two consecutive *revisions* of a single artifact being edited. It was also noted that, in general, transformations may take place in spaces with multiple dimensions including revisions, replicas, languages, and features [11,31]. Exploring *bx* in these multidimensional spaces was proposed as an important area for future work.

State-based vs. operation-based. Many of the projects presented at the meeting focused on using *bx* to propagate changes made to the source or target of the transformation. These update translators fall into two distinct categories. In *state-based* approaches, the update translator operates on the source and target structures themselves. For example, to translate an update made to the target back to the source, the translator takes as an argument the post-update state of the target (as well as the original state of the source) and calculates an appropriately-modified source [15,16,41,42]. By contrast, in *operation-based* approaches, updates are expressed in a transformation language, and the update translator propagates the updates themselves through the transformation. In either case, update translation produces instances that are consistent with respect to the *bx* [81,102].

Properties of bx Several different properties of *bx* were discussed at the meeting. The properties of the relation that captures when the source and target are consistent—e.g., whether the relation is an injective source-to-target function or a total target-to-source function or both, or even if the relation is a function in either direction at all—impact *bx* in fundamental ways. Most systems stipulate that the transformations a *bx* comprises must obey the *definitional properties* of *correctness* and *hippocraticness* [97] (these properties roughly correspond to PutGet and GetPut laws in the lens framework [42]; for details of this correspondence, see elswhere [30,99]), with *undoability* [97] (corresponding to PutPut for lenses [42]) as an optional property. The *reversibility* of a transformation was discussed as an example of an *operational property*: not only must the inverse exist, it must also be computable efficiently. Lastly, properties such as *incrementality* [46], *minimality of changes* [19,44], and *preservation of recent changes* were classified as *quality properties*. Although these properties have intuitive appeal, they are not well understood formally and are an important area for further study.

Constructing bx Several different techniques for constructing *bx* were discussed at the meeting. There are three main approaches. In the first approach, the user programs the forward transformation and the backward transformation is obtained (almost) for free, because the forward transformation either was built from smaller primitives that are bidirectional (e.g., [78,101,102]) or can be made bidirectional (e.g., [76,108]). The second approach, used in lenses, is similar but subtly different: the programmer specifies the backward transformation and obtains the forward transformation for free (e.g., [16, 42]). In the third approach, the user defines a *mapping*—i.e., the specification of the relation that captures when the source and target are consistent—and the forward and backward transformations are derived from the mapping automatically, possibly with additional manual refinement (e.g., [6,81,93]). A related topic is the composition of bidirectional translators, e.g., lenses and synchronizers, into larger systems ([30,31,41]).

4 Towards a BX Benchmark

The GRACE-BX meeting clarified the need to have effective criteria for comparing different *bx* approaches and assessing progress in the field. Benchmarks are widely used in academia and industry as a framework for comparing competing approaches and technologies. Insightful examples of benchmark suites exist for scenarios with cross-discipline application, including graph transformation [105], query and transformation of XML [91], generic programming (query and transformation of tree-shaped data) [90], and schema mappings [1].

Accordingly, the meeting included a discussion session on a potential benchmark suite for bidirectional transformations, codenamed BXBenchmark. The following goals of BXBenchmark were identified:

– Provide a *platform* for comparing expressiveness, usability, and efficiency of different *bx* approaches and technologies.
– Catalog proven *bx scenarios* and interesting variations thereof (e.g., the ubiquitous classes-to-tables mapping and variations thereof).

– Collect *implementations* of *bx* scenarios.
– Clarify the relevance of the scenarios across different approaches and technologies and thereby *reveal commonalities and differences.*

Organization of the Benchmark Suite

To help picturing the envisaged benchmark suite BXBenchmark, we sketch its possible organization. At the top level, the suite is organized by a collection of *scenarios*. Each scenario comprises components as described below. (We use the ubiquitous classes-to-tables mapping as a running example.)

Name. For instance: "Classes to Tables".

Rationale. For instance: "Cover a baseline scenario of model-driven development with just a few modeling concepts for classes in the sense of the OO paradigm and tables in the sense of the relational paradigm".

Metadata. Based on an appropriate *taxonomy* of bidirectional transformations, the scenarios are to be semantically annotated. To provide a simple example, scenarios would be annotated as being graph- vs. string- vs. tree-based. (The scenario of the running example is graph-based.) We discuss some elements of an emerging taxonomy shortly.

Sample data. Each scenario will be accompanied by a collection of types (or classes, schemas, regular expressions, or other relevant *metamodel* information) that model sample data for the scenario, as well as actual conformant *sample data.* In addition, generators and mappers for metamodels and sample data may provide access to sample data across platform, format, and programming paradigm while meeting constraints on size or others.

Configurations. The term "configuration" is used here as a proxy for comprehensively describing the execution of a bidirectional transformation including sample data (inputs and results), a description of updates and auxiliary parameters such as size measures when data is generated.

Specialization. Scenarios may be related by specialization. For instance, we may think of "Classes to Tables" as an abstract scenario with fundamentally different O/R mapping options as concrete specializations.

Any scenario can now be implemented by any number of implementations using particular technologies. For instance, one technology may address the scenario by describing both directions of the *bx* scenario separately, whereas another technology may leverage a designated transformation language for *bx.* We also need some way of documenting the assumed difficulty, feasibility, completeness, or infeasibility of a specific implementation (option). To this end, the role of metadata, as announced above, will be extended.

An Emerging Taxonomy for Metadata

We expect this taxonomy to cover properties of *bx* that are useful in documenting (by means of metadata) commonalities and differences between the different scenarios, the different implementations thereof, the different *bx* approaches and technologies. A few candidate properties are illustrated:

Injectivity. Given a scenario, is the basic forward transformation injective? If it is not injective, what other properties are possibly exploited to obtain a useful backward transformation?

Kinds of changes. Given a scenario or a *bx* approach, what kinds of changes are needed or supported? For instance, one can distinguish updates vs. insertion vs. deletion. When focusing on updates, one can distinguish structured vs. primitive updates.

Synchronization orientation. Given an implementation of a scenario or a technology, is synchronization involved? If so, how can we further classify and qualify the form of synchronization at hand? (For instance, do we face synchronization based on a constraint mechanism?)

Semantics preservation. Given a scenario that involves a sort of (programming) language as the domain of the sources related by *bx*, does the *bx* promise or require semantics preservation? (See the scenario "For-loop Desugaring" presented shortly.) Other programming language-related notions may be similarly leveraged to contribute properties for taxonomy, e.g., type preservation and dependency on flow analysis.

Some Candidate Scenarios for BXBenchmark

"Roman/Arabic Number Conversion". This is a trivial example which can be used to provide a basic demonstration of any transformation technology. The conversions are bijective and can be reasonably represented by a pair of unidirectional transformations.

"Add Index to Address Book" (inspired by [55]). Given a simple collection of addresses of persons (say represented as an XML tree), an index for the names of the persons is added by the forward transformation. The redundancy created by the extra name index triggers update challenges.

"Classes to Tables". This scenario (described earlier) is a baseline scenario of model-driven development. Various technologies have readily addressed some variant of it. It appears to be promising to try organizing all these variants. Conceptually, it is an (O/R) mapping example which hence involves two paradigms: the OO paradigm and the relational paradigm. Object models and relational schemas may both be represented as graphs (with edges for associations or key constraints). In fact, whereas a tree-based approach suffices for the previous scenario, a graph-based approach may be more appropriate for the present scenario. (We view "Classes to Tables" as a representative of the larger class of relatively direct metamodel transformations: WSDL to/from EMF, BPMN to/from BPEL, EMF to/from XSD, UML sequence to/from communication diagrams, etc.)

"Collapse/Expand State Diagrams". Starting from a hierarchical state diagram (involving some nesting), a flat view is to be provided, and any modifications on the flat view should be reflected eventually in the hierarchical view. The basic flattening transformation is non-injective, and hence, the view complement must be taken into account when mapping back the flattened and possibly updated state diagram to a nested one.

"For-loop Desugaring". Given a tiny (imperative) programming language, provide a syntax desugaring transformation, e.g., the translation of for-loops to while loops. The desugared syntax is further subjected to a refactoring transformation. The bidirectional transformation challenge is to be able to revert desugaring even past refactoring. Just like the previous scenario, desugaring is non-injective. The specific contribution of the scenario lies in its well-understood interaction with programming language types and semantics. Ideally, one would want to establish, by construction, that desugaring and its reversion are type- and semantics-preserving. (A related but distinctive scenario, inspired by [28,74], is the preservation of whitespace and comments by transformations that abstract from whitespace and comments.)

"Round-trip Engineering for Java Applet Models and Code" (inspired by [2,3]). In the forward direction, JApplet framework use is expressed at the modeling level in terms of a feature model, subject to a code-generating interpretation of the model. In the backward direction, framework use is extracted from code by queries and presented as feature models. Updates may be performed both on code and feature model. This scenario stands out with its involvement of two rather distant abstraction levels: imperative OO code vs. more grammatical and declarative feature models. This scenario is also relatively challenging in that it requires flow analysis.

"Textual and Graphical Program Editor". Given (a subset of) a Java/C#-like language, provide a capability for simultaneous editing in textual and graphical mode. It is assumed that the graphical mode provides a limited view on programs. For instance, it may be limited to types and relationships. This scenario involves "Text to Model" and "Model to Text" components, and thereby involves specific challenges such as layout preservation (of both textual and graphical layout) and comment preservation (where we assume that comments only appear in the textual representation). Further, this scenario requires the ability to incrementally (locally) change the graphical view in reply to local changes to the textual view, and vice versa.

"Reversible FFT" (inspired by [72,116]). A Fast Fourier Transform is an efficient algorithm to compute the Discrete Fourier Transform and its inverse. Compared to the above scenarios, this scenario specifically involves a computationally expensive problem.

Status and Future Prospects

An open source project has been created to host future efforts on BXBenchmark.[3] The project is actively seeking contributors. Designated workshops or bird-of-a-feather sessions and hackathons may be scheduled to make progress on the suite. Work on the benchmark suite is likely to feature prominently at any follow-up meetings.

[3] `https://sourceforge.net/projects/bxbenchmark/`

5 Conclusion

Bidirectional transformation is a field of interest that spans many sub-disciplines of computer science. The GRACE-BX meeting served as a useful checkpoint to match capabilities and research efforts, and to examine differences in approaches and assumptions. The references cited in this report constitute the beginnings of a comprehensive bibliography of *bx*-related work across the sub-disciplines. The new benchmark suite will continue the collaboration effort by allowing researchers with different research aims to contribute solutions to common problems. In addition to the emerging benchmark, we will continue to foster collaboration between communities by applying for a Dagstuhl seminar. We may also hold workshops at conferences as part of the effort to establish the benchmark.

Acknowledgments. We would like to thank all the participants of the GRACE-BX meeting for their contributions at the meeting and their input during the preparation of this report: Holger Bock Axelsen, Jean Bezivin, Jácome Cunha, Davide Di Ruscio, Zinovy Diskin, Bernd Fischer, Kathleen Fisher, Soichiro Hidaka, Robert Glück, Hiroyuki Kato, David Lutterkort, Kazutaka Matsuda, Shin-Cheng Mu, Alfonso Pierantonio, Keisuke Nakano, Ali Razavi, Hui Song, Antonio Vallecillo, Janis Voigtläender, Yingfei Xiong, Meng Wang, and Andrzej Wasowski.

References

1. Alexe, B., Tan, W.-C., Velegrakis, Y.: STBenchmark: towards a benchmark for mapping systems. In: Proceedings of the VLDB conference, vol. 1(1), pp. 230–244 (2008), http://www.stbenchmark.org/
2. Antkiewicz, M.: Framework-Specific Modeling Languages. Ph.D thesis, University of Waterloo, Electrical and Computer Engineering (2008)
3. Antkiewicz, M., Bartolomei, T.T., Czarnecki, K.: Automatic extraction of framework-specific models from framework-based application code. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering, pp. 214–223. ACM, New York (2007)
4. Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with Round-Trip Engineering. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 692–706. Springer, Heidelberg (2006)
5. Antkiewicz, M., Czarnecki, K.: Design Space of Heterogeneous Synchronization. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 3–46. Springer, Heidelberg (2008)
6. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of Framework-Specific Modeling Languages. IEEE Transactions on Software Engineering (2009) (to appear)
7. Arenas, M., Libkin, L.: XML data exchange: Consistency and query answering. Journal of the ACM 55(2) (2008)
8. Arenas, M., Pérez, J., Riveros, C.: The recovery of a schema mapping: bringing exchanged data back. In: PODS 2008: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 13–22. ACM, New York (2008)

9. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible Machine Code and Its Abstract Processor Architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007)
10. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Transactions on Database Systems (TODS) 6(4), 557–575 (1981)
11. Batory, D.S., Azanza, M., Saraiva, J.: The objects and arrows of computational design. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 1–20. Springer, Heidelberg (2008)
12. Bennet, C.H.: Logical Reversibility of Computation. IBM Journal of Research and Development 17(6), 525–532 (1973)
13. Benton, N.: Embedded interpreters. Journal of Functional Programming 15(4), 503–542 (2005)
14. Berdaguer, P., Cunha, A., Pacheco, H., Visser, J.: Coupled Schema Transformation and Data Conversion for XML and SQL. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 290–304. Springer, Heidelberg (2006)
15. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: Proceedings of ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2008), January 2008, pp. 407–419 (2008)
16. Bohannon, A., Pierce, B., Vaughan, J.: Relational lenses: a language for updatable views. In: PODS 2006: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 338–347. ACM, New York (2006)
17. Brabrand, C., Møller, A., Schwartzbach, M.I.: Dual Syntax for XML Languages. Information Systems 33(4-5), 385–406 (2008); Short version in DBPL 2005 (2005)
18. Buneman, P., Fernandez, M.F., Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion. VLDB Journal: Very Large Data Bases 9(1), 76–110 (2000)
19. Buneman, P., Khanna, S., Tan, W.-C.: On propagation of deletions and annotations through views. In: PODS 2002: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 150–158. ACM, New York (2002)
20. Castro, P., Melnik, S., OAdya, A.: ADO.NET entity framework: raising the level of abstraction in data programming. In: SIGMOD 2007: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 1070–1072. ACM, New York (2007)
21. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, Proceedings, pp. 222–231. IEEE Computer Society, Los Alamitos (2008)
22. Cleve, A., Hainaut, J.-L.: Co-transformations in Database Applications Evolution. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 409–421. Springer, Heidelberg (2006)
23. Culpepper, R., Felleisen, M.: Debugging macros. In: GPCE 2007: Proceedings of the 6th international conference on Generative programming and component engineering, pp. 135–144. ACM, New York (2007)
24. Cunha, J., Saraiva, J., Visser, J.: From spreadsheets to relational databases and back. In: PEPM 2009: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, pp. 179–188. ACM, New York (2008)

25. Czarnecki, K., Helsen, S.: Classification Of Model Transformation Approaches. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture (2003), http://www.softmetaware.com/oopsla2003/czarnecki.pdf
26. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
27. Dayal, U., Bernstein, P.A.: On the Correct Translation of Update Operations on Relational Views. ACM Transactions on Database Systems (TODS) 7(3), 381–416 (1982)
28. de Vanter, M.L.V.: Preserving the Documentary Structure of Source Code in Language-Based Transformation Tools. In: 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), Proceedings, pp. 133–143. IEEE Computer Society, Los Alamitos (2001)
29. Diskin, Z.: Mathematics of generic specifications for model management. In: Rivero, Doorn, Ferraggine (eds.) Encyclopedia of Database Technologies and Applications, pp. 351–366. Idea Group (2005)
30. Diskin, Z.: Algebraic Models for Bidirectional Model Synchronization. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 21–36. Springer, Heidelberg (2008)
31. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: algebraic foundations and the tile notation. In: Comparison and versioning of software models. 3rd Int. Workshop affiliated with ICSE 2009 (2009); To appear in IEEE Digital Library
32. Eger, D.T.: Bit Level Types (2005) (unpublished manuscript), http://www.yak.net/random/blt/blt-drafts/03/blt.pdf
33. Ehrig, Engels, Kreowski, Rozenberg (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1. World Scientific Publishing, Singapore (1997)
34. Ehrig, Engels, Kreowski, Rozenberg (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2. World Scientific Publishing, Singapore (1999)
35. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006)
36. Eramo, R., Pierantonio, A., Romero, J., Vallecillo, A.: Change Management in Multi-Viewpoint Systems using ASP. In: Proceedings of 5th International Workshop on ODP for Enterprise Computing (WODPEC 2008) (September 2008)
37. Fagin, R.: Inverting schema mappings. ACM Transactions on Database Systems (TODS) 32(4) (2007)
38. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theoretical Computer Science 336(1), 89–124 (2005)
39. Fisher, K., Gruber, R.: PADS: a domain-specific language for processing ad hoc data. In: PLDI 2005: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 295–304. ACM, New York (2005)
40. Fisher, K., Walker, D., Zhu, K., White, P.: From dirt to shovels: fully automatic tool generation from ad hoc data. In: POPL 2008: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 421–434. ACM, New York (2008)

41. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Exploiting Schemas in Data Synchronization. Journal of Computer and System Sciences 73(4) (June 2007); Short version in DBPL 2005 (2005)
42. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(3) (2007)
43. Foster, J.N., Pilkiewcz, A., Pierce, B.C.: Quotient lenses. In: ICFP 2008: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, pp. 383–396. ACM, New York (2008)
44. Fuxman, A., Kolaitis, P.G., Miller, R.J., Tan, W.C.: Peer data exchange. ACM Transactions on Database Systems (TODS) 31(4), 1454–1498 (2006)
45. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1), 21–43 (2009)
46. Green, T.J., Ives, Z.G., Tannen, V.: Reconcilable differences. In: ICDT 2009: Proceedings of the 12th International Conference on Database Theory, Proceedings, pp. 212–224. ACM, New York (2009)
47. Green, T.J., Karvounarakis, G., Ives, Z.G., Tannen, V.: Update Exchange with Mappings and Provenance. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007, pp. 675–686. ACM, New York (2007)
48. Hainaut, J.-L., Tonneau, C., Joris, M., Chandelon, M.: Schema Transformation Techniques for Database Reverse Engineering. In: Elmasri, R.A., Kouramajian, V., Thalheim, B. (eds.) ER 1993. LNCS, vol. 823, pp. 364–375. Springer, Heidelberg (1994)
49. Halevy, A.Y., Ives, Z.G., Suciu, D., Tatarinov, I.: Schema Mediation in Peer Data Management Systems. In: Proceedings of the 19th International Conference on Data Engineering, ICDE 2003, pp. 505–516. IEEE Computer Society, Los Alamitos (2003)
50. Harren, M., Raghavachari, M., Shmueli, O., Burke, M., Bordawekar, R., Pechtchanski, I., Sarkar, V.: XJ: facilitating XML processing in Java. In: WWW 2005: Proceedings of the 14th international conference on World Wide Web, pp. 278–287. ACM, New York (2005)
51. Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)
52. Hidaka, S., Hu, Z., Kato, H., Nakano, K.: A Compositional Approach to Bidirectional Model Transformation. In: New Ideas and Emerging Results Track of 31st International Conference on Software Engineering (ICSE 2009, NIER Track) (May 2009) (to appear)
53. Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Towards Compositional Approach to Model Transformation for Software Development. In: 24th Annual ACM Symposium on Applied Computing (March 2009)
54. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: PEPM 2004: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 178–189. ACM, New York (2004); See [55] for a journal version
55. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. Higher-Order and Symbolic Computation 21(1-2), 89–118 (2008); See [54] for a short version

56. Huang, G., Mei, H., Yang, F.: Runtime recovery and manipulation of software architecture of component-based systems. Automated Software Engineering 13(2), 257–281 (2006)
57. Karvounarakis, G., Ives, Z.G.: Bidirectional Mappings for Data and Update Exchange. In: Proceedings of 11th International Workshop on the Web and Databases, WebDB 2008 (2008), http://webdb2008.como.polimi.it/images/stories/WebDB2008/paper35.pdf
58. Kawanaka, S., Hosoya, H.: biXid: a bidirectional transformation language for XML. In: Proceedings of ICFP 2006: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, pp. 201–214. ACM, New York (2006)
59. Keller, A.M.: The Role of Semantics in Translating View Updates. Computer 19(1), 63–73 (1986)
60. Keller, A.M., Jensen, R., Agarwal, S.: Persistence software: bridging object-oriented programming and relational databases. In: SIGMOD 1993: Proceedings of the 1993 ACM SIGMOD international conference on Management of data, pp. 523–528. ACM, New York (1993)
61. Königs, A., Schürr, A.: MDI - a Rule-Based Multi-Document and Tool Integration Approach. Journal of Software and System Modeling 5(4), 349–368 (2006); Special Section on Model-based Tool Integration
62. Krishnamurthi, S., Erlich, Y.-D., Felleisen, M.: Expressing Structural Properties as Language Constructs. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 258–272. Springer, Heidelberg (1999)
63. Lämmel, R.: Grammar Adaptation. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001)
64. Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: First International Workshop on Software Evolution Transformations, November 2004, 5 pages (2004), http://homepages.cwi.nl/~ralf/CoupledSoftwareTransformations/
65. Lämmel, R.: LINQ to XSD. In: Proceedings, PLAN-X 2007, Programming Language Technologies for XML, An ACM SIGPLAN Workshop collocated with POPL 2007, pp. 95–96 (2007), http://www.plan-x-2007.org/plan-x-2007.pdf
66. Lämmel, R.: Style normalization for canonical X-to-O mappings. In: PEPM 2007: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 31–40. ACM, New York (2007)
67. Lämmel, R., Lohmann, W.: Format Evolution. In: Proceedings of 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001). books@ocg.at, vol. 155, pp. 113–134. OCG (2001)
68. Lämmel, R., Meijer, E.: Mappings Make Data Processing Go 'Round. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 169–218. Springer, Heidelberg (2006)
69. Lämmel, R., Meijer, E.: Revealing the X/O impedance mismatch (Changing lead into gold). In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 285–367. Springer, Heidelberg (2007)
70. Lämmel, R., Zaytsev, V.: An Introduction to Grammar Convergence. In: Integrated Formal Methods, 7th International Conference, IFM 2009, Proceedings. LNCS, vol. 5423, pp. 246–260. Springer, Heidelberg (2009)
71. Lee, H., Antkiewicz, M., Czarnecki, K.: Towards a Generic Infrastructure for Framework-specific Integrated Development Environment Extensions. In: 2nd International Workshop on Domain-Specific Program Development (DSPD), in association with GPCE (2008), http://www.labri.fr/perso/reveille/DSPD/2008

72. Li, J.: Low noise reversible MDCT (RMDCT) and its application in progressive-to-lossless embedded audio coding. IEEE Transactions on Signal Processing 53(5), 1870–1880 (2005)
73. Liquid, X.: http://www.liquid-technologies.com/
74. Lohmann, W., Riedewald, G.: Towards Automatical Migration of Transformation Rules after Grammar Extension. In: 7th European Conference on Software Maintenance and Reengineering (CSMR 2003), Proceedings, pp. 30–39. IEEE Computer Society, Los Alamitos (2003)
75. Lutterkort, D.: Augeas–A Configuration API. In: Proceedings of the Linux Symposium, Ottawa, ON, July 2008, pp. 47–56 (2008)
76. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ICFP 2007: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, pp. 47–58. ACM, New York (2007)
77. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalizing Programs with Duplication through Complementary Function Derivation. JSSST Journal: Computer Software 26(2), 5 (2009) (to appear) (in Japanese)
78. McBrien, P., Poulovassilis, A.: Data Integration by Bi-Directional Schema Transformation Rules. In: 19th International Conference on Data Engineering, ICDE 2003, Proceedings, pp. 227–238. IEEE Computer Society, Los Alamitos (2003)
79. Meertens, L.: Designing Constraint Maintainers for User Interaction (June 1998) (manuscript), http://www.kestrel.edu/home/people/meertens
80. Meijer, E., Beckman, B., Bierman, G.: LINQ: reconciling object, relations and XML in the .NET framework. In: SIGMOD 2006: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 706–706. ACM, New York (2006)
81. Melnik, S., Adya, A., Bernstein, P.: Compiling mappings to bridge applications and databases. In: SIGMOD 2007: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 461–472. ACM, New York (2007)
82. Miller, R., Hernández, M., Haas, L., Yan, L., Ho, C., Fagin, R., Popa, L.: The Clio Project: Managing Heterogeneity. ACM SIGMOD Record 30(1), 78–83 (2001)
83. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. In: PLDI 2007: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 146–155. ACM, New York (2007)
84. Mu, S.-C., Hu, Z., Takeichi, M.: An Algebraic Approach to Bi-directional Updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
85. Nakano, K., Hu, Z., Takeichi, M.: Consistent Web Site Updating based on Bidirectional Transformation. In: 10th IEEE International Symposium on Web Site Evolution (WSE 2008) (October 2008)
86. Oliveira, J.: Transforming Data By Calculation. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 134–195. Springer, Heidelberg (2008)
87. Pratt, T.W.: Pair Grammars, Graph Languages and String-to-Graph Translations. Journal of Computer and System Sciences 5, 560–595 (1971)
88. Ramsey, N.: Embedding an interpreted language using higher-order functions and types. In: IVME 2003: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators, pp. 6–14. ACM, New York (2003)

89. Reynolds, J.: Types, abstraction and parametric polymorphism. In: Information Processing 1983, Proceedings, pp. 513–523. Elsevier, Amsterdam (1983)
90. Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., Oliveira, B.: Comparing libraries for generic programming in Haskell. In: Haskell 2008: Proceedings of the first ACM SIGPLAN symposium on Haskell, pp. 111–122. ACM, New York (2008)
91. Schmidt, A., Waas, F., Kersten, M., Florescu, D., Manolescu, I., Carey, M., Busse, R.: The XML Benchmark Project. Technical report, CWI, Amsterdam, The Netherlands, Technical Report INS-R0103 (April 2001), http://www.xml-benchmark.org/
92. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
93. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903. Springer, Heidelberg (1995)
94. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
95. Shu, N.C., Housel, B.C., Taylor, R.W., Ghosh, S.P., Lum, V.Y.: EXPRESS: A Data EXtraction, Processing, amd REStructuring System. ACM Transactions on Database Systems (TODS) 2(2), 134–174 (1977)
96. Song, H., Xiong, Y., Hu, Z., Huang, G., Mei, H.: A Model-Driven Framework for Constructing Runtime Architecture Infrastructures. Technical report, National Institute of Informatics, Japan, Technical Report GRACE-TR-2008-05 (December 2008), http://grace-center.jp/en/rsc_tr.html
97. Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
98. Stevens, P.: A Landscape of Bidirectional Model Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
99. Stevens, P.: Towards an Algebraic Theory of Bidirectional Transformations. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 1–17. Springer, Heidelberg (2008)
100. Takahashi, S., Matsuoka, S., Miyashita, K., Hosobe, H., Kamada, T.: A Constraint-Based Approach for Visualization and Animation. Constraints 3(1), 61–86 (1998)
101. Terwilliger, J., Delcambre, L., Logan, J.: Querying through a user interface. Data & Knowledge Engineering 63(3), 774–794 (2007)
102. Terwilliger, J.F.: Graphical User Interfaces as Updatable Views. Ph.D thesis, Portland State University (2009)
103. Terwilliger, J.F., Melnik, S., Bernstein, P.A.: Language-integrated querying of XML data in SQL server. PVLDB 1(2), 1396–1399 (2008)
104. Vallecillo, A.: A Journey through the Secret Life of Models. In: Model Engineering of Complex Systems (MECS). Dagstuhl Seminar Proceedings, vol. 08331 (2008), http://drops.dagstuhl.de/opus/volltexte/2008/1601

105. Varró, G., Schürr, A., Varro, D.: Benchmarking for Graph Transformation. In: VLHCC 2005: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 79–88. IEEE Computer Society, Los Alamitos (2005)
106. Vermolen, S., Visser, E.: Heterogeneous Coupled Evolution of Software Languages. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 630–644. Springer, Heidelberg (2008)
107. Visser, J.: Coupled Transformation of Schemas, Documents, Queries, and Constraints. ENTCS 200(3), 3–23 (2008)
108. Voigtländer, J.: Bidirectionalization for free! (Pearl). In: POPL 2009: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 165–176. ACM, New York (2009)
109. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
110. Wadler, P.: Views: a way for pattern matching to cohabit with data abstraction. In: POPL 1987: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 307–313. ACM, New York (1987)
111. Wadler, P.: Theorems for free! In: FPCA 1989: Proceedings of the fourth international conference on Functional programming languages and computer architecture, pp. 347–359. ACM, New York (1989)
112. Wang, M., Gibbons, J.: Translucent Abstraction: Safe Views through Bidirectional Transformation (2008), http://www.comlab.ox.ac.uk/files/711/Bidi.pdf
113. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 164–173. ACM, New York (2007)
114. Xiong, Y., Zhao, H., Hu, Z., Takeichi, M., Song, H., Mei, H.: Beanbag: Operation-based Synchronization with Intra-relations. Technical Report GRACE-TR-2008-04, Center for Global Research in Advanced Software Science and Engineering, National Institute of Informat iontics, Japan (December 2008), http://grace-center.jp/downloads/GRACE-TR-2008-04.pdf
115. XML Beans, http://xmlbeans.apache.org/
116. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a Reversible Programming Language. In: CF 2008: Conference on Computing Frontiers, Proceedings, pp. 43–54. ACM, New York (2008)
117. Yokoyama, T., Axelsen, H.B., Glück, R.: Reversible Flowchart Languages and the Structured Reversible Program Theorem. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 258–270. Springer, Heidelberg (2008)

Author Index

Albert, Patrick 249
Apel, Sven 4

Bastarrica, María Cecilia 197
Baudry, Benoit 148
Behring, Alexander 132
Bergmann, Gábor 20
Bertrand, Frédéric 116
Bézivin, Jean 229, 249
Bollati, Verónica Andrea 181
Brunelière, Hugo 197

Cicchetti, Antonio 35
Cuadrado, Jesús Sánchez 229
Czarnecki, Krzysztof 260

de Lara, Juan 83
Del Fabro, Marcos Didonet 249
Di Ruscio, Davide 35

Foster, J. Nathan 260
Fuentes-Fernández, Rubén 52

García-Magariño, Iván 52
García Molina, Jesús 229
Gómez-Sanz, Jorge J. 52
Grønmo, Roy 67
Guerra, Esther 83

Hettel, Thomas 100
Hillairet, Guillaume 116
Horváth, Ákos 20
Hu, Zhenjiang 213, 260

Janda, Florian 4
Jouault, Frédéric 197, 229, 249

Kästner, Christian 4
Krogdahl, Stein 67

Lafaye, Jean Yves 116
Lämmel, Ralf 260
Lange, Christian F.J. 239
Lawley, Michael 100

Marcos, Esperanza 181
Mottu, Jean-Marie 148
Møller-Pedersen, Birger 67
Mühlhäuser, Max 132

Orejas, Fernando 83

Petter, Andreas 132
Pierantonio, Alfonso 35
Pierce, Benjamin C. 1

Ráth, István 20
Raymond, Kerry 100

Schürr, Andy 260
Sen, Sagar 148
Song, Hui 213
Stevens, Perdita 165

Takeichi, Masato 213
Terwilliger, James F. 260
Trujillo, Salvador 4

van Amstel, Marcel F. 239
van den Brand, Mark G.J. 239
Vara, Juan Manuel 181
Varró, Dániel 20
Vela, Belén 181
Vignaga, Andrés 197

Xiong, Yingfei 213