Designing Round-Trip Systems by Change Propagation and Model Partitioning

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Mirko Seifert
geboren am 01.12.1978 in Rodewisch

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)

Prof. Dr. Jean Bézivin
(INRIA - École des Mines de Nantes, France)

Tag der Verteidigung: Dresden, den 28. Juni 2011

Dresden im April 2011

Confirmation

I confirm that I independently prepared this thesis with the title *Designing Round-Trip Systems by Change Propagation and Model Partitioning* and that I used only the references and auxiliary means indicated in the thesis.

Dresden, April 6, 2011

Dipl.-Inf. Mirko Seifert

Abstract

Software development processes incorporate a variety of different artifacts (e.g., source code, models, and documentation). For multiple reasons the data that is contained in these artifacts does expose some degree of redundancy. Ensuring global consistency across artifacts during all stages in the development of software systems is required, because inconsistent artifacts can yield to failures. Ensuring consistency can be either achieved by reducing the amount of redundancy or by synchronizing the information that is shared across multiple artifacts. The discipline of software engineering that addresses these problems is called Round-Trip Engineering (RTE).

In this thesis we present a conceptual framework for the design RTE systems. This framework delivers precise definitions for essential terms in the context of RTE and a process that can be used to address new RTE applications. The main idea of the framework is to partition models into parts that require synchronization—*skeletons*—and parts that do not—*clothings*. Once such a partitioning is obtained, the relations between the elements of the skeletons determine whether a deterministic RTE system can be built. If not, manual decisions may be required by developers.

Based on this conceptual framework, two concrete approaches to RTE are presented. The first one—*Backpropagation-based RTE*—employs change translation, traceability and synchronization fitness functions to allow for synchronization of artifacts that are connected by non-injective transformations. The second approach—*Role-based Tool Integration*—provides means to avoid redundancy. To do so, a novel tool design method that relies on role modeling is presented. Tool integration is then performed by the creation of role bindings between role models.

In addition to the two concrete approaches to RTE, which form the main contributions of the thesis, we investigate the creation of bridges between technical spaces. We consider these bridges as an essential prerequisite for performing logical synchronization between artifacts. Also, the feasibility of semantic web technologies is a subject of the thesis, because the specification of synchronization rules was identified as a blocking factor during our problem analysis.

The thesis is complemented by an evaluation of all presented RTE approaches in different scenarios. Based on this evaluation, the strengths and weaknesses of the approaches are identified. Also, the practical feasibility of our approaches is confirmed w.r.t. the presented RTE applications.

Acknowledgment

The route that led to this thesis has been very long and without the continuous support of many people over many years I certainly would not have been able to get to this point. Even though I might end up forgetting someone, I will try to acknowledge the people I am most thankful for.

The starting point for becoming a computer scientist was set by my grandparents Inge and Hilmar who gave in to the endless begging of their ten-year-old grandson and spent a fortune on one of the first home computers. I am very grateful for being given this opportunity and thank them for putting so much trust in me. Shortly after, a friend of mine—Henning Liebold—gave me a book about programming. I took the book home and read it front to back until I understood all these magical things. If I remember correctly, this book enormously increased my interest in programming languages. For nostalgic reasons, I do still keep the book. Thank you, Henning!

Then, many years later I met one of my today's colleagues—Florian Heidenreich. The exact circumstances of our first talk cannot be recapitulated, but for what we know, Florian mentioned that there is an open PhD position in the group I joined shortly after. I still consider this as a very fateful moment.

In addition to Florian, many others shared my days at the Software Technology Group. First, there was Steffen Zschaler from whom I learned a lot about scientific writing and working. Then, other very important and inspiring people joined the group. In particular Jendrik Johannes and Christian Wende became not only highly valued colleagues, but also close friends. I can say that I am most thankful for all the thoughts, coffees and beers that we have shared over the past five years. Together with Florian, the two of you can be considered the most influential people w.r.t. the work I have performed during my PhD. You have been great, because you have shared a wonderful attitude toward what you do and because you have always provided an atmosphere I will definitely miss. In addition, I am very thankful to Birgit Grammel, Julia Schroeter, Jan Reimann and Sebastian Götz for commenting on draft versions of this thesis.

I would also like to thank my supervisor Prof. Uwe Aßmann. He did always provide me with countless pointers to the work of brilliant researchers, helping me to base my work on their great insights. He draw connections between things within my work and to the outside, I did not see myself. I remember certain moments when he made me realize what my thesis was actually about, which I am very thankful for. In addition, Uwe has always been a source of enthusiasm. Working with people who care that much

about what they do was basically great. Also, I am most thankful to him for taking the time to comment on the draft versions of this thesis.

I highly appreciate that Prof. Jean Bézivin has accepted to be my external reviewer. I consider this as a great honor, because I have always looked up to the excellent research of him and his group. Also, I would like to thank my second internal reviewer Prof. Alexander Schill for taking the time to comment on my thesis.

While doing my PhD, our secretary Katrin assisted me with booking flights, finding hotel rooms, filling out administrative forms, reserving rooms, and most importantly, ordering coffee-related goods. For all this, I am also very thankful.

During my time at the university, many students had to suffer under my supervision. Accompanying, reading and grading their thesis' has been a great experience. I have learned a lot during this work. Also, some of my students turned out to be quite talented and parts of this thesis are even based on their work. I cannot mention all their names here, but I must mention Jan Reimann, being one of my last students. Winning the Springer Best Paper Award at the MODELS conference was one of the greatest moments and I have never been that proud of one of my students before. It is great to know that you are on the team now.

In addition to the people influencing my research from close distance, I am also very thankful to all the fellow computer scientists in the world, who had to review one of my papers or listen to one of my talks. I do not even know all their names, but I have often received valuable feedback on my work.

I am also thankful to all the organizations who funded my work within the various research projects I took part in. Without this financial support, I would have never been able to conduct my research, trying to find answers to so many interesting problems.

Besides the great amount of support that I have received at work, I would like to thank my parents both for their financial and moral support. I am also deeply grateful for having such a great brother. He contributed to this thesis by providing healthy distractions whenever we had lunch together. I would like to extend my thanks to my mother in law—Angelika—for taking care of my daughter, especially during these last days of my thesis. In addition, I am thankful for all my friends who never got bored hearing about my thesis.

Finally, I am deeply indebted to my girlfriend Denise and our daughter Lene. Definitely, these two suffered the most while I wrote up this thesis. During the last weeks before submission, my nerves were all on edge and I am extremely thankful for their support and tolerance. You have always managed both to provide me the encouragement to continue with this thesis, but at the same time you brought to my attention that there are more important things in life than synchronizing models. I love you.

Publications

This thesis is partially based on the following peer-reviewed publications:

- Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert and Christian Wende: *Derivation and Refinement of Textual Syntax for Models.* In Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009), 23-26th June 2009, Enschede, The Netherlands.
- Mirko Seifert: *Opportunities and Challenges of Traceable Graph Rewriting Systems.* In Proceedings of the 3rd International Workshop on Graph and Model Transformation (GraMoT 2008), Satellite event of the 30th International Conference on Software Engineering 2008 (ICSE 2008), 12th May 2008, Leipzig, Germany.
- Jendrik Johannes, Roland Samlaus and Mirko Seifert: *Round-trip Support for Invasive Software Composition Systems.* In Proceedings of the International Conference on Software Composition 2009 (SC 2009), 2-3rd July 2009, Zurich, Switzerland.
- Federico Rieckhof, Mirko Seifert and Uwe Aßmann: *Ontology-based Model Synchronisation.* In Proceedings of the 3rd Workshop on Transforming and Weaving OWL Ontologies in MDE/MDA (TWOMDE 2010), 30th June 2010, at TOOLS 2010 federated conferences, Malaga, Spain.
- Mirko Seifert, Christian Wende and Uwe Aßmann: *Anticipating Unanticipated Tool Interoperability using Role Models.* In Proceedings of the 1st Workshop on Model Driven Interoperability (MDI'2010) (co-located with MODELS 2010), 5th October 2010, Oslo, Norway.
- Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende: *Closing the Gap between Modelling and Java.* Tool demonstration at the 2nd International Conference on Software Language Engineering (SLE'09), 5-6th October 2009, Denver, Colorado.
- Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende: *JaMoPP: The Java Model Parser and Printer.* Technical Report, Technische Universität Dresden, Fakultät Informatik, TUD-FI09-10 September 2009, ISSN 1430-211X.

– Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende and Marcel Böhme: *Generating Safe Template Languages.* In Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE'09), 4-5th October 2009, Denver, Colorado.

– Jan Reimann, Mirko Seifert and Uwe Aßmann: *Role-based Generic Model Refactoring.* In Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), 6-8th October 2010, Oslo, Norway.

The following peer-reviewed publications cover work that is closely related to the content of the thesis, but not contained herein:

– Mirko Seifert and Roland Samlaus: *Static Source Code Analysis using OCL.* In Proceedings of the workshop OCL Tools: From Implementation to Evaluation and Comparison (OCL 2008), Satellite event of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008), 30th September 2008, Toulouse, France.

– Mirko Seifert and Christian Werner: *Specification of Triple Graph Grammar Rules using Textual Concrete Syntax.* In Proceedings of the 7th International Fujaba Days, 16-17th November 2009, Eindhoven University of Technology, The Netherlands.

– Mirko Seifert and Stefan Katscher: *Debugging Triple Graph Grammar-based Model Transformations.* In Proceedings of 6th International Fujaba Days, 18-19th September 2008, Dresden, Germany.

– Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende: *Construct to Reconstruct - Reverse Engineering Java Code with JaMoPP.* In Proceedings of the International Workshop on Reverse Engineering Models from Software Artifacts (R.E.M.'09), 15th October 2009, Lille, France.

Contents

1 Introduction

1.1 Motivation

Within the last decades, software has become an indispensable part of our lives. Sometimes we make use of it in a direct and visible fashion, for example, when using a spreadsheet calculator to plan a project budget. At other times, we barely notice the presence of software even though it might be much more crucial (e.g., while being on an airplane). Despite its intangible nature, todays wealth of the global society heavily depends on the correctness and reliability of software systems. From a current perspective, this trend is evermore increasing. More software is deployed in an increasing number of places, performing more and more safety-related tasks. Besides the increasing amount of software that is used in total, the interconnection between software systems has also grown. Often the term *system of systems* is used to emphasize this progression.

During all stages in the development of such software systems (i.e., specification, design, implementation, test, deployment, and maintenance) one particular criterion of quality is of utter importance—consistency. Various rules of thumb reflect this need for consistency (e.g., the Don't Repeat Yourself (DRY) principle). The reason for the strong emphasis on consistency is that inconsistent artifacts can easily cause serious problems. For example, consider a system where the table names used in a relational database schema do not match the names used in queries that are sent to the database. As a result of this inconsistency, the running system will eventually expose failures.

Having said this, it must be admitted, that actual software projects often violate the consistency principle. Software artifacts are duplicated, source code is copied in order to reuse it, transformations repeat information by translating it to other formats, or

data is spread across multiple artifacts. Consequently, lots of redundancy is introduced. Obviously keeping track of all these redundant bits of data is difficult, for a considerable large software system even impossible to do without sophisticated machinery.

It is important to note that introducing redundancy is not necessarily caused by the ignorance or inattention of software engineers. Often there are good reasons to do so. For example, when the analysis of a large amount of data is not practicable because of performance reasons, it is reasonable to reduce the data set by extracting the relevant information only and running the analysis on the reduced (copied) subset. In other cases tools might not be able to operate on the same data representation. Transforming data to apply essential tools is therefore also a rational reason to replicate information.

These two important aspects of software engineering—ensuring consistency among artifacts and the inevitable introduction of redundancy—open an area of conflict. While one struggles to achieve global consistency, practical constraints force oneself to compromise, always increasing the risk of failures lurking into a redundant system description. To reduce these risks, the redundancies that are willingly introduced while engineering software, must be tracked and changes need to be synchronized automatically.

With the advent of new development paradigms such as Model-Driven Software Development (MDSD) [1] and Domain-Specific Languages (DSLs) [2], ensuring consistency across all software artifacts has become even more important. These new methods provide excellent conceptual means to address large-scale systems. However, before MDSD can be applied to its full extent to solve practical problems, many open research questions need to be addressed. For example, growing systems over multiple stages starting at high-level, abstract system descriptions is a great technique to address complexity. But, practical implementations of the idea have been reported to be not mature enough to keep up with this high expectations [3]. In particular the integration of different tools and modeling techniques has been observed to be problematic [3]. Also, model transformations, being at the heart of MDSD, create an enormous amount of redundancy as artifacts are duplicated, which in turn renders MDSD processes to be complicated. Dealing with these redundant models is a challenging, but equally important task.

A summary of MDSD-related research questions has been presented by Robert France and Bernhard Rumpe in [4]. Managing models, which includes preserving consistency, is one important point on this research road map. To benefit from the core ideas of MDSD to their full extent, methods are required that provide support to handle or avoid redundancy and to allow for easy integration of development tools. Besides the consistency preservation problems that are related to MDSD, earlier work [5] has identified the need to provide developers with sophisticated support during the resolution of inconsistencies as a main research topic in software engineering.

Today, the term Round-Trip Engineering (RTE) is used to denote methods that keep dependent models consistent whenever changes are applied. Without sophisticated support for RTE, modern MDSD processes will always suffer from replication and disintegration, and what was meant to solve complex problems, will become a complex problem

itself. To increase the practical feasibility of MDSD, the goal of our work is to provide novel approaches to RTE, which ease the automatic synchronization of software artifacts and allow to integrate different technologies to foster their individual strengths.

The research conducted in this thesis is based on the following main ideas:

- systematic partitioning and composition of software models,
- strict separation of technical and logical synchronization aspects,
- deterministic and indeterministic change propagation.

Using these ideas as the foundations for the approaches developed in this thesis, some of the problems found in an analysis of the field of RTE are addressed. The goal of this work is therefore to provide means to manage redundancy in MDSD processes.

1.2 Scope

The scope of this thesis is limited to the automation and simplification of dealing with redundancies in MDSD. While the results of this work may be applicable to other artifacts that are not related to software development, the focus of this work is strongly oriented at software development artifacts. We do also limit the applicability to artifacts which can be represented by the use of object-oriented models (i.e., graph structures). Although we belief that the majority of artifacts can be represented this way, establishing such a representation is a prerequisite. This thesis does not target synchronization across metalevels such as the co-evolution of metamodels and models. Also, we do not consider distributed development environments.

1.3 Contributions

The main contributions of this thesis are a conceptual framework to build RTE systems and two novel, generic approaches to model synchronization that instantiate the concepts of this framework is different ways. Based on a detailed analysis of the diverse problems which cause redundancy in MDSD, the main issues are identified in Chap. 3. Each of the two approaches addresses one or more of these problems. In addition, we investigate on the use of semantic web technologies for the specification of consistency rules as the creation of such specifications has been identified as a problem itself.

The conceptual framework is presented in Chap. 4 and delivers a set of terms and definitions that are required to build *RTE systems*. The framework is based on two main ideas. First, the strict separation of technical and logical synchronization aspects. Second, the partitioning and composition of models. Both ideas provide the conceptual foundation for the subsequent chapters. Also, a process for the design of RTE systems

based on the aforementioned principles is given, together with some basic architectural patterns that frequently reoccur.

The first approach—backpropagation-based RTE—is presented in Chap. 6 and resolves some of the limitations that can be found when using bidirectional model transformations for synchronization. For example, scenarios where inverse transformations cannot be derived automatically or do not even exist, can be tackled. Also, we do explicitly support non-injective transformations.

The second approach—role-based RTE (cf. Chap. 8)—targets the reduction of redundant data by explicitly designing tools for interoperability. To do so, the data model required by tools to operate on is cleanly separated from the physical representation of this data. Minimal effort is required to implement the proposed tool design and future problems caused by redundant data are completely avoided.

In Chap. 7, semantic web technologies are evaluated w.r.t. their feasibility in synchronization scenarios. The aim of this work is to make use of the formal semantics that ontologies are based on. In contrast to Chap. 6 and 8 that present generic RTE approaches, ontology-based model synchronization is more specific and not as extensive as our two main approaches.

In summary, the thesis provides means to tackle existing redundancy in scenarios which were not addressable by existing approaches, or to ease the creation of RTE systems. It also shows how the problem of synchronization can drastically be reduced by appropriate tool modeling and design.

1.4 Organization

This thesis is organized as follows. First, required background knowledge is briefly presented in Chap. 2. Then, an analysis about the sources of redundancy and the problems that arise while trying to preserve consistency is conducted in Chap. 3. Based on this analysis, the most challenging problems are identified. Bridging technical spaces, being one of these problems, is tackled in Chap. 5—mostly by using existing approaches. Then, the three main chapters of the thesis, which rely on the bridges that are presented in Chap. 5 follow. These form the main contributions of this thesis.

First, in Chap. 6 synchronization based on asymmetric backpropagation is presented. This approach explicitly deals with existing redundancy. That is, it provides means to synchronize related software artifacts without requiring to change existing tools. Thereafter, we employ the formal semantics of ontologies in Chap. 7 to synchronize redundant data. The third approach is different as it tries to avoid redundancy instead of dealing with it after it is created. Based on role models, different tools are bound to the same physical representation of data (Chap. 8). All three main approaches are evaluated in Chap. 9, a comparison with existing work is conducted in Chap. 10. The thesis is then concluded by a summary and a collection of future work in Chap. 11.

1.5 Terms and Definitions

Unless stated otherwise, the following informal definitions of general terms are used in this thesis.

- An *artifact* is a set of data that follows a given schema. We assume that any artifact can be represented as a model.
- A *model* is an artifact that conforms to a metamodel based on the metalanguage Essential Meta Object Facility (EMOF) [6]. As such it can also be considered as a typed, attributed graph.
- A *specification* is a model.
- A *document* is an artifact without executable semantics.
- A *program* is an artifact that can be executed. If there is metamodel for the language the program is written in, the program is also a model.
- A *domain* is a set of related concepts.
- The term *ontology* refers to an Web Ontology Language (OWL) [7] ontology.
- A *model transformation* is the process of computing a set of output models from a set of input models.
- A *transformation specification* is a specification, which can be used to control a model transformation.
- A *source model* is the input for a model transformation.
- A *target model* is the output of a model transformation.
- A *model fragment* is a partial model (i.e., a subset of a model).

1.6 Typographical Conventions

Throughout this thesis the following conventions are used:

- New terms appear in *italic font* once they are introduced.
- Names of metaclasses and their properties are written in `typewriter font`. Metaclass names start with upper-case letters, whereas properties start with a lower-case character.
- Source code and textual models are also written in `typewriter font`.

- URLs are typeset in `typewriter font`.
- Inline quotations use *italic font*.
- All other quotations appear in

  separate, indented paragraphs.

2
Background

This chapter briefly recapitulates the foundations that are required to understand the work presented in the subsequent parts of the thesis. The first three subsections contain relevant terms and concepts to model domain knowledge. Section 2.1 starts with an overview of object-oriented modeling techniques. For the scope of this thesis, a particular variation of object-oriented modeling, namely role modeling, is most important and therefore explained separately in Sect. 2.2. An alternative approach to create and use conceptual models are ontologies and Description Logics (DL). The foundations for both will be presented in Sect. 2.3.

Once models have been created in one way or the other, their relations need to be studied to understand how consistency among artifacts can be achieved. When using models as primary artifact type in development processes, model transformations are the natural choice to express relations between models and to derive models from each other. However, there are various ways to transform models, both conceptually and technically. Section 2.4 will give an overview of these.

If models expose redundancy, methods to manage duplicate data are required. We recapitulate general goals of such model synchronization methods in Sect. 2.5. In particular, the notion of consistency across models is considered in this section. Concrete methods for model synchronization require a thorough understanding the relations between models. For example, tracing information, which can be gathered during the execution of a transformation can provide details about redundant parts of models. Therefore, Sect. 2.6 gives details about different traceability approaches.

2.1 Modeling and Metamodeling

Modeling is a term that can be found on many different occasions and which is given various meanings depending on the context it is used in. In particular engineers use the term frequently. Therefore, this section will start with common definitions of what a model is, what kinds of models exist and how models can be formalized (Sect. 2.1.1). The latter does in particular demand for descriptions of models (i.e., models of models), which are also called *metamodels* and presented in Sect. 2.1.2. Then, languages that are used to express models will be discussed in Sect. 2.1.3, before Sect. 2.1.4 sheds some light on the storage and management of models.

2.1.1 Models

To understand the nature of modeling it is important to realize that models are intrinsic instruments used by humans for all kinds of cognitive processes. The psychologist Michael S. Gazzaniga being a notable researcher in cognitive neuroscience argues that the human mind creates a *mental model* of our environment, which is then used to make decisions or render new thoughts [8]. The same view has been held by Ludwig [9] in the context of software engineering. Software development does require substantial cognitive performance and is therefore often concerned with the creation of models. However, these models do not need to be mental only, but must be persisted during all phases of software development to allow their mechanical processing and analysis.

The properties of such models being a natural means to describe real or imaginary worlds have been summarized in [10]. One may shortly list them as *made for a purpose*, *incomplete*, and *selected by purpose*. Similarly, Stachowiak [11] uses the three criteria *pragmatism*, *reduction* and *mapping* to distinguish models from their originals. The first two properties coincide and are also the most important ones. Every model is an abstraction of its counterpart in the world that is subject to modeling. Consequently only details that are relevant to a specific purpose will be included in the respective model. All other aspects are left out. A model is therefore incomplete with respect to the modeled subject, but should be complete with regard to its purpose.

Models can be divided into descriptive and prescriptive ones [12]. The former type is used to capture reality, thus describing the modeled subject. The objective is to model things the way they are. The latter type specifies the reality (i.e., subjects are constructed from prescriptive models). Both types are used in software engineering. Descriptive models are employed when analyzing or re-engineering software, while prescriptive ones are used to specify software systems during forward engineering [13].

The concrete appearance of a model can be quite different. Models can be hierarchical or have a graph structure. As for any other kind of information various ways to structure a model do exist. However, not all possible kinds of models are equally well suited for a particular purpose. Even more, not all types of models are as easy to be understood by

humans as others. For example, the amount of facts a human can process simultaneously is limited. Therefore, we tend to split information into consumable units. This leads to hierarchical models, because they allow to decompose complicated structures.

To create models that can be processed mechanically, the way to build valid models needs to be determined by some formalism. Usually, mathematical structures (e.g., trees or graphs) are employed to provide such a basis for modeling. In the scope of this thesis, we will restrict ourselves to models that are represented by typed, attributed graphs [14]. These graphs, being an essential part of object-oriented programming have gained wide acceptance over the last decades. Thus, they are extensively used to create models of the real or imaginary world. To be more specific, we will stick to models that conform to the EMOF standard [6].

The EMOF specification defines *composite properties* (cf. [6] Sect. 12.5), which yields a distinct spanning tree if we assume that models are connected graphs. Full connectivity is not enforced by the EMOF standard, but can always by realized by introducing an artificial root node for all models. This root node can connect all previously disconnected parts of the model. In the scope of this thesis we will assume connected graphs or require the introduction of such a root node.

2.1.2 Metamodels and Metalayers

If we consider different types of models (e.g., process models, data models or behavioral models), the question about the formalization of a model comes to mind. Such a formalization is required, because models shall be used to model software systems. To do so, models need to be processed by machines, which requires a formal description of what a model is. In other words: How can the structure of a model be specified?

In the previous section, we have selected EMOF as a formalism to create and describe models. However, this restriction is far from useful when it comes to capturing the knowledge of a specific domain. Instead of using nodes, edges and arbitrary generic types to represent a domain, more specific models are needed. That is, when modeling within a certain domain, additional restrictions (e.g., predefined types) are required. Naturally, we can use models to express these restrictions—we model models.

The process of modeling models is called *metamodeling*. Here, a metamodel is used to specify the structure of other models. Often this is referred to as *a model conforming to a metamodel*. Essentially, the metamodel restricts the set of valid models, which is allowed by the general formalism—in our case the set of all EMOF models. One can also consider metamodels as constraints that models must fulfill to be well-formed. Therefore, metamodels are means to define local consistency rules for models.

Having seen that machines require metamodels to process models one may continue to describe one layer of models using another one, creating a stack of metalayers. In principle this is possible, but there is good reason to limit the number of layers. To understand why this limitation is needed, let us consider the reason for introducing

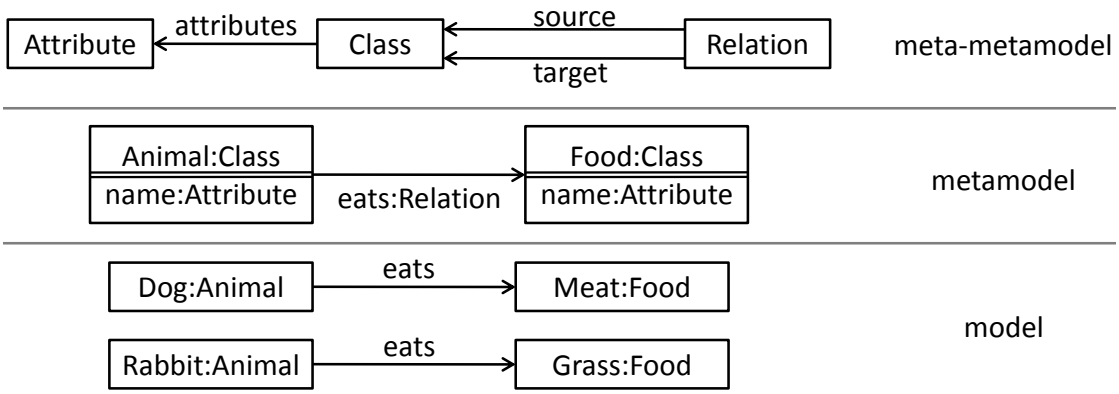


Figure 2.1: Three simplified metalayers built using object-oriented modeling.

metamodels in the first place. Metamodels were required to have a clear specification of the set of possible models. Having a finite representation of a possibly infinite number of models allows machines to process models. The model specification (i.e., the metamodel) allows to handle arbitrary models even though their concrete shape is not known yet.

When tools share the same metamodel, they are basically able to process the same set of models.[1] If one wants to handle different metamodels with a single tool, a third metalayer is needed. This layer specifies what metamodels look like. To reduce the set of metalayers to the minimum, the third layer must be designed such that it can be used to specify many different kinds of metamodels. In the object-oriented modeling paradigm, the concepts *class*, *relation* and *attribute* are used to build such widely applicable metamodels. Using these concepts, metamodels for various purposes can be specified. In Fig. 2.1, simple example models spread across three metalayers are shown.

On the top layer some concepts of object-oriented modeling (class, relation and attribute) are shown. The middle layers uses these concepts to model animals and food they eat. The lower layer uses concepts from the middle layer to model concrete animals and particular kinds of food. Because each layer uses (i.e., *instantiates*) concepts from the upper one, we say that model elements on two adjacent metalayers are connected by an implicit `instanceOf` relation[2]. Also, on the level of entire models, this relation is denoted as `conformsTo` [15]. In the example shown in Fig. 2.1, the `instanceOf` relation holds between `Animal` and `Class`, `Food` and `Class`, `eats` and `Relation`, `Dog` and `Animal`.

In principle three layers are sufficient to allow processing of models by machines [16]. Two layers are not sufficient, because one cannot give a single metamodel that is usable to describe substantially different domains. But, even though metamodels can be quite different, a third layer is sufficient to describe all of them using the same concepts.

[1] This assumes that a concrete physical representation of metamodel concepts is given
[2] In [15], Def. 2 the `instanceOf` relation is denoted as function μ

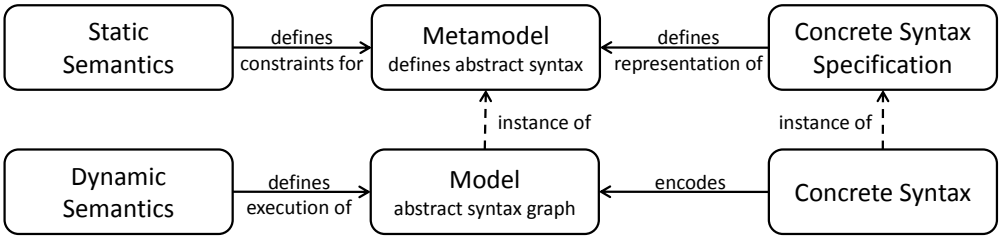


Figure 2.2: Components of modeling languages.

The Object Management Group (OMG), being a prominent advocate of modeling languages and techniques, promoted a four layer architecture to align their metamodels. One may wonder why some of these undoubtedly important standard modeling languages rely on four layers instead of three. A detailed discussion of this issue is out of scope of this thesis, but to shortly resolve this discrepancy it must be noted that the OMG uses an explicit real world level. The layer `M0` reflects objects from the real world [6]. The OMG Meta Object Facility (MOF) [6] standard is not restricted to four layers, but can be used with any number of layers greater than or equal to two (cf. [6] Sect. 7.2). More thorough insights about the implications of this approach can be found in [17]. This includes a detailed comparison of different possible metamodel hierarchies.

Another fact to be pointed out here, is the need to define meta-metamodels recursively. When looking at the simplified object-oriented meta-metamodel shown in Fig. 2.1, one can observe that the top layer can be modeled using its own concepts. `Class`, `Attribute` and `Relation` are classes on their own, while `source`, `target` and `attributes` are relations. The reason for using a recursive model on the meta-metalayer is basically to avoid using another, different description mechanism. One may also consider this more a pragmatic reason rather than a conceptual one [18]. The models on this layer must simply describe themselves to eliminate the usage of another metalanguage.

2.1.3 Modeling Languages

A *modeling language* is a formal language that can be used to express abstractions of the real or imaginary world (i.e., it allows to phrase models). It can also be used to specify abstractions of models (i.e., models of models). Conceptually, a modeling language allows to specify a set of valid models. The definition of a modeling language must therefore define this set formally. To do so, a metamodel is used. In addition, modeling languages usually do impose a specific way to express members from this set and, to be useful, they assign meaning to models. Thus, every modeling language consists of the components that are depicted in Fig. 2.2.

First, a metamodel defines the concepts and relations that are used by models expressed in the language. Second, a concrete syntax specification defines how the concepts

and relations (i.e., the abstract syntax) are expressed in terms of a perceptible representation. Common examples include textual syntax, which uses characters to formulate models, or graphical syntax, which adds geometrical shapes to symbolize model elements and relations among them. The specification of concrete syntax can be performed in different ways. Grammars can be used to specify textual syntax [19]. For graphical syntax other definition mechanisms can be used. Depending on the concrete application of a language, one type of syntax can be more suitable than another. Generally speaking, both have advantages and disadvantages, which can be compensated by using multiple kinds of syntax in parallel. Also, though rather unusual in software engineering, other types of syntax, such as haptic or audible syntax can be used to express models as well.

Third, each modeling language is accompanied by a semantics definition. This definition entails both a static and a dynamic part. Both definitions assign meaning to the concepts defined by the metamodel. The static semantics places constraints on the set of valid models that are not captured by the metamodel. For example, Object Constraint Language (OCL) [20] constraints can be used to restrict this set. In contrast, the dynamic semantics specifies the execution of models. For the purpose of modeling software systems, dynamic semantics is often defined by mapping concepts to concepts of another modeling language which does have a well-established dynamic semantics. Compilers and model transformations are well-known examples for such mappings.

When looking at existing modeling languages, different kinds of languages can be found. Some languages are more generic than others. For example, the Unified Modeling Language (UML) [21, 22] can be used for a variety of software engineering tasks. To name a few, UML allows to model data structures, behavior or use cases for software applications. The same applies to the Systems Modeling Language (SysML) [23]. Other modeling languages are specifically bound to a certain task. An Entity Relationship Diagram (ERD) [24] is tailored to specify relational data, but not very useful for other tasks. The list of existing modeling languages is large and enumerating even a fraction of them is sheer impossible. Unfortunately, to the best of our knowledge, no extensive classification of modeling languages is available. There are some classifications which cover languages for specific domains [25, 26, 27], but a general one could not be found.

A special category of languages are the metamodeling languages. These languages allow the specification of modeling languages, being a modeling language on their own. For example, MOF is the metamodeling language used to define UML. Other metamodeling languages are Ecore [28], Kernel Meta Meta Model (KM3) [15] and MEMO [29].

2.1.4 Model Repositories

Model repositories are software systems that persist and manage models. This task is similar to filing other software artifacts, but does also impose some differences [30]. The main purpose of model repositories is to provide access for tools, which operate on serialized models. Serialization can be performed in various ways. Model repositories

can employ relational databases to store models or use version control systems to persist them. Object-oriented databases can also be employed in this context. Examples of model repositories are the Modelbus project [31] and Connected Data Objects (CDO).

To access models, repositories usually provide an Application Programming Interface (API) (e.g., Java Metadata Interface (JMI) [32]). Like databases, repositories can also expose models or model elements by means of a query language. Similar to the large number of existing modeling languages there are also many query languages to choose from. OCL is probably the most well-known language. Epsilon Object Language (EOL) [33] is an alternative.

When models are stored in repositories one must keep track of their relations. For example, if one model was derived using a model transformation (cf. Sect. 2.4) this information should be kept in the repository. To do so, *megamodels* were introduced [34] and implemented in [35].

Megamodels represent the relations among models and model transformations. This information (i.e., the megamodel) can then be used to reason about the referenced models. For example, if a model acting as the source for one or more transformations is changed, these transformations can be triggered by the use of the megamodel. It must be noted here, that megamodels do not necessarily need to be part of a model repository. They can also be used in other contexts, even if models are not persisted at all. But, the other way around, model repositories pose a great need for megamodels.

2.2 Role Modeling

Having seen how models can be built using classes, relations and attributes in the previous section, this section explains a variant of the object-oriented modeling paradigm—*role modeling*. After a conceptual overview in Sect. 2.2.1, the implementation of roles and their embedding in the previous modeling approach will be discussed in Sect. 2.2.2.

2.2.1 The Role Concept

On a conceptual level, roles can be understood as the relations in which entities can take part [10, 36, 37]. Instead of modeling the relations between objects individually for each class—as done in pure object-oriented modeling—roles are an explicit, first-class concept to model relations. Therefore, roles are also denoted as *role types*.

The term *role* was originally defined in [36] as *a part of a relationship that is played by an object*. Later on, Kristensen gave a similar definition in [37]:

> A role of an object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects.

In the following, we will differentiate between *roles* and *role types*. The latter term refers to the definition of a relationship, while the former is a concrete instance of such

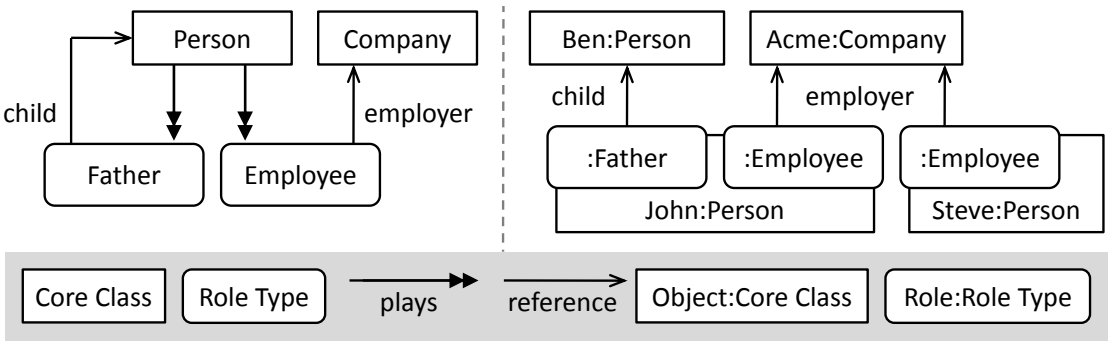


Figure 2.3: Example role model (left) and a corresponding instance (right).

a relation. For example, a role type `Employee` can define a relation to a company, which is played by multiple persons. The concrete relation that holds between one person and a company is the denoted as the `Employee` role of the respective person.

Role types essentially model interactions between objects. In contrast to entities, roles do not have an identity on their own, but are rather identified, by the entities that *play a role*. Multiple objects can participate in one role and one role type can be played by multiple objects. If an entity object plays a role, the role is considered to *be attached* to this object. Over the lifetime of an entity, it can gather new roles or loose existing ones.

In a complementary fashion, the concept of an entity class (or *core class*) has been defined in [38] as:

> A class of a core object is a set of properties and behavior which are intrinsic meaning that they are rarely modified for use in various application contexts, and so basically include no properties and behavior on interactions.

A specification of role types, classes, relations and constraints between role types is called a *role model* [10]. To give a simple example for a role model and to illustrate the difference between roles, role types, objects and core classes, consider Fig. 2.3.

Here, both `Person` and `Company` are entities that have a unique identity. The role `Father` and `Employee` do not have such an identity and must therefore be played by entity objects. The role type `Father` models a relation between two persons, namely a father and his child. The identity of a fatherhood is fully determined by the two persons involved in this relationship. Similarly, the role type `Employee` connects a person and a company as long as a person is working for a particular enterprise.

Figure 2.3 indicates that the distinction between objects having a well defined identity and role types modeling the relations between these objects is a very natural one. Even though relations can also be modeled using standard object-oriented modeling techniques, the notion of role types as first-class concept provides a distinction that reflects the conceptual difference between objects bearing identity and those which do not. This

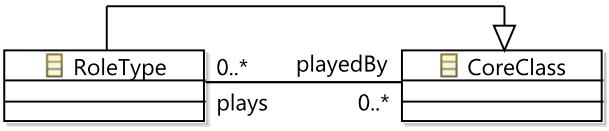


Figure 2.4: Metamodel for role types.

distinction can also be found in [39], where it is denoted as *rigid* and *non-rigid* types. In the remainder of this thesis the rigid types will be called *core classes*.

If expressed in terms of a metamodel, role types form a new class, which is related to classes having an identity (i.e., core classes). This relation is bidirectional. Roles are played by objects, whereas objects play roles. This relation is depicted in Fig. 2.4.

At a closer look, role types share many commonalities with core classes. Obviously, they can carry data (i.e., have attributes) and define operations. Clients that want to access a role do use exactly the operations specified in the respective role type. Conceptually, role types are special classes which can only be distinguished based on their semantics. Whether a class does have an identity or not is determined by the actual objects—either real or imaginary—it reflects. As a consequence, the metamodel shown in Fig. 2.4 contains a generalization relation between role types and core classes.

This generalization relation does also allow roles to be played by other roles. This way, a stack of roles is introduced. For example, the roles of type `Employee` (cf. Fig. 2.3) can play a new role `Taxpayer`, which relates employees to their annual tax declaration. This new role cannot be played by persons, since the calculation of taxes depends on the income, which is a property of the `Employee` role.

When using roles to model a domain, restrictions—*role constraints*—can apply. Role types can exclude each other, forcing objects to play either a role of one type or the other, but not both at the same time. For example, a person cannot play the roles `Father` and `Mother` simultaneously. This kind of constraint is called *role-prohibited* [40]. Besides this prohibition of playing two roles at the same time, Riehle mentions the following other constraint types *role-dontcare*, *role-implied*, *role-equivalent*.

The first one (role-dontcare) states that two role types are independent from each other. The second (role-implied) can be used to express that playing a role of the first type implies the need for playing another role of the second type, too. The last constraint type (role-equivalent) equalizes two role types—playing a role of one type is equivalent to playing a role of another type. In addition to these four basic constraint types, more complex constraints can apply to relations between objects (i.e., to roles). One can find various kinds of invariants specified over object relations in [41].

Once a role model has been filled with a set of role types and core classes, *role bindings* are required to specify how a particular role is played by objects of a core class. For example, if the role type `Father` provides an operation `care()`, a specification is needed

to determine how a `Person` implements this operation. Therefore, different role players can behave differently even though they might play the same role. The role binding defines the behavior of objects of a given class within a certain context (i.e., the context of playing a particular role). This process is also called *role composition*, because the properties and the behavior of role types and core classes are composed. We will shortly see more details about this in Sect. 2.2.2.

To summarize, role types and role models are an extension to object-oriented modeling (cf. [42] for an extension of UML). The particular strength of role modeling is having a first-class distinction between parts (i.e., properties and behavior) of an object that are inherently bound its identity and parts that are relevant in certain contexts (i.e., while interacting with other objects) only. Role modeling allows to separate these two kinds of properties. It does also include the ability to dynamically compose and decompose object structures by attaching and detaching roles to (or respectively from) objects.

2.2.2 Mapping Roles to Object-oriented Models

To put role modeling into practice, there is basically two options. First, one can translate role models to a General Purpose Language (GPL) that supports roles as a first-class concept. Examples for such languages are Self [43], ObjectTeams [44], LasagneJ [45], PowerJava [46], EpsilonJ [47] and the Java Role Package [48]. Despite the maturity of some of these languages, object-oriented technology (e.g., languages and tools) still dominates software development in our days. Thus, a second option is to map role models to pure object-oriented models. This option is interesting both from a practical point of view—to use role modeling in existing development processes—and from a conceptual perspective—to understand the implications of role modeling more closely.

The first pattern to implement role types with classic object-oriented technology is the *role object pattern* [49]. This pattern decorates core classes with role types to bind role players to the roles they play. A simplified class diagram of this pattern is shown in Fig. 2.5. The interface for the core class (`ComponentCore`) is captured by an abstract class `Component`. Both the core class (`ComponentCore`) and the role type (`ComponentRole`) extend this interface. This allows clients to access core objects that are decorated with role objects equivalent to objects which are not decorated. The role type provides the same interface as the core class and delegates methods calls to its core object. Thus, the behavior of the core class is preserved even in the presence of attached role objects. Moreover, role types can expose additional behavior or state.

The essence of the role object pattern are the two relations `roles` and `core` which connect the role and the core type. Without the former reference, the pattern is equivalent to the decorator pattern [50]. However, the `roles` reference allows to navigate from core objects to their roles, which is essential to obtain objects in one context and use them in another. In addition to the very basic form of the role object pattern shown in Fig. 2.5, there are plenty of other variants which can be found in the literature [51, 52].

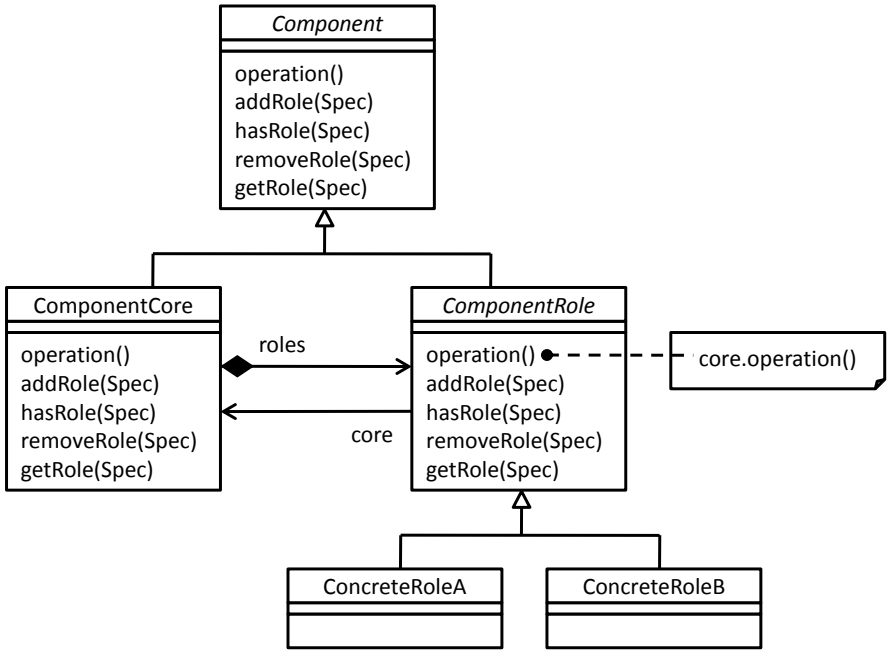


Figure 2.5: Simplified class diagram for the role object pattern (from [49]).

The major drawback of using a mapping to object-oriented technology instead of using a dedicated GPL with support for roles, is the *object schizophrenia problem* [53]. This problem is caused by the fact that each object carries its own identity. If roles are represented by separate objects, as it is the case when using the role object pattern, each role has an individual identity. These identities are different from the identity of the respective core object. Clients relying on the identity of objects can therefore observe different identities for the same core object depending on the context it is used in.

2.3 Ontologies

Ontologies are rooted in the field of knowledge representation. More precisely, the main goal of ontologies is to capture knowledge about a specific domain in a formal and precise way. An ontology represents knowledge about the concepts that can be found in a domain. Relations between concepts can be declared, such as two concepts being equal to each other or one concept being a generalization of a second one. Furthermore, instances of concepts can be given (i.e., concrete objects that embody the concept).

Besides the formalization of explicit knowledge about a domain, ontologies allow to derive implicit knowledge from explicit facts. This process is called *reasoning* and can make use both of the defined facts and the rules that are defined for a particular domain.

The ontologies presented in this thesis are based on a particular kind of logic that is presented subsequently in Sect. 2.3.1. The concrete language used to represent ontologies is OWL, which is explained in Sect. 2.3.2. As we will also use Semantic Web Rule Language (SWRL) [54], a brief introduction to this language can be found in Sect. 2.3.3.

2.3.1 Description Logics

Description Logics (DL) are a set of related formal languages that can be used to represent knowledge [55]. Depending on the set of operators that are allowed to express knowledge, different languages can be obtained. And, according to the complexity of these operators, each language can expose different behavior w.r.t. the efficiency of associated decision procedures.

DL have a formal syntax and semantics and therefore provide a valid mathematical grounding for ontologies. Central terms in DL are *concepts*, *roles*, *individuals* and *relations*. Similar to the distinction between models and metamodels presented in Sect. 2.1.2, DL distinguish between the *TBox*—the terminological box—and the *ABox*—the assertion box. Statements about relations between concepts belong to the TBox, whereas sentences that involve relations involving individuals account for the ABox.

In contrast to object-oriented modeling, DL do not rely on the Unique Name Assumption (UNA). That is, if two concepts carry different names, they can still represent the same thing. For example, two concepts `Node` and `Element` can denote exactly the same set of individuals. In contrast, two classes in object-oriented modeling that carry different names, model two distinct sets of objects.

Also, DL are based on the Open World Assumption (OWA), which states that knowledge which is missing, does not imply that the negation of the missing fact is true. For example, if no information is available about the fact whether two nodes are connected or not, one cannot imply that there is no connection between the two. Again, this is substantially different from object-oriented modeling, where the absence of facts implies that the fact is false. For example, the missing connection between nodes does inevitably represent the fact that there is no connection in object-oriented modeling.

As DL are based on logics, one cannot only represent knowledge using concepts, individuals, roles and relations, but derive implicit facts by drawing logical conclusions. For example, if an individual `x` is an instance of a particular concept `A` and a second concept `B` is declared to be equivalent to `A`, reasoners for DL are able to derive that `x` is also an instance of `B`, even though this information was not given explicitly.

When performing such reasoning, the computational properties of the different members of the DL family are of utter importance. Depending on the concrete DL language that is employed, the knowledge base and the requested deduction, deriving such implicit knowledge can expose different complexity. For many variants of DL, decision procedures and respective algorithmic complexities are known, which allows to make assumptions about the time and memory required to reason about certain kinds of knowledge.

2.3.2 Web Ontology Language (OWL)

Web Ontology Language (OWL) [7] is the most prominent ambassador of ontologies and became quite popular with the advent of the semantic web [56]. It can therefore also be used in conjunction with the Resource Description Framework (RDF) [57]. Historically, OWL 1 came in three variants—OWL-Lite, which had the least expressiveness, OWL-DL, which was directly based on DL and OWL-Full, which made no restrictions on expressiveness (i.e., equivalent to RDF). Consequently, OWL-Full can pose problems with w.r.t. the computational complexity of decision procedures.

In OWL 2—the current version of OWL—there are also three language fragments, which are now called *OWL profiles*. The profiles are named *OWL 2 EL*, *OWL 2 QL*, and *OWL 2 RL*. Detailed information about the restrictions of each profile and the resulting computational complexity can be found in [58].

To create OWL ontologies, different concrete syntax can be used. The OWL standard [7] mentions five options. The *RDF/XML* syntax is required by tools to adhere to the standard. Optional syntax variants are *OWL/XML*, *Functional Syntax*, *Manchester Syntax* [59] and *Turtle*. Since, the RDF/XML syntax is rather difficult to read, more human-friendly syntax like the OWL Manchester Syntax are quite popular. An except from an OWL ontology written in Manchester Syntax can be found in Listing 2.1.

```
1 Class: VegetarianPizza
2   EquivalentTo: Pizza and not (hasTopping some FishTopping) and not (hasTopping some MeatTopping)
3   DisjointWith: NonVegetarianPizza
```

Listing 2.1: Excerpt of an OWL ontology in Manchester Syntax (from [59]).

The semantics of OWL 2 is specified in [60] and based on the $\mathcal{SHROIQ}$ [61] variant of DL. In comparison, the semantics of OWL-DL was based on $\mathcal{SHOIN}$, which is a subset of $\mathcal{SHROIQ}$.

2.3.3 Semantic Web Rule Language (SWRL)

SWRL is an extension of OWL that enables the use of Horn-like rules in combination with the OWL knowledge base. Within this thesis, we will restrict ourselves to the DL-safe subset of SWRL. SWRL rules consist of two parts—the antecedent and the consequent—and can be read as follows. If all atoms in the antecedent are true, then the consequent must also be true. The antecedent and the consequent consist of a set of atoms, which can be used to check whether a variable or a concept: a) adheres to some description, b) is within a data range, c) fulfills a property, d) is the same as or different from another variable or concept, or e) fulfills some built-in predicate.

A simplified metamodel of SWRL can be found in Fig. 2.6. One can spot the different kinds of atoms, as well as the metaclass `Rule` with its two parts—`Antecedent` and `Consequent`. The part of the metamodel that is used to define variables or to refer to existing OWL metaclasses is not shown here.

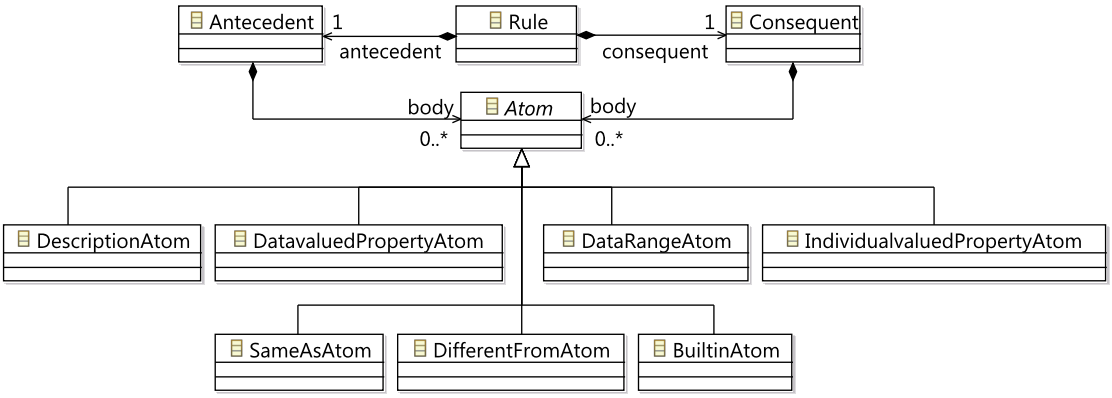


Figure 2.6: Simplified metamodel of SWRL.

The SWRL standard defines two concrete syntaxes for rules. One is based on the Extensible Markup Language (XML) [62] representation of OWL. The other one is human readable. Listing 2.2 shows two variants of the same rule using the two types of concrete syntax. The example is taken from [54].

```
1  // human readable
2  hasParent(?x1,?x2) /\ hasBrother(?x2,?x3) => hasUncle(?x1,?x3)
3
4  // same rule in XML syntax
5  <ruleml:imp>
6    <ruleml:_rlab ruleml:href="#example1"/>
7    <ruleml:_body>
8      <swrlx:individualPropertyAtom swrlx:property="hasParent">
9        <ruleml:var>x1</ruleml:var>
10       <ruleml:var>x2</ruleml:var>
11     </swrlx:individualPropertyAtom>
12     <swrlx:individualPropertyAtom swrlx:property="hasBrother">
13       <ruleml:var>x2</ruleml:var>
14       <ruleml:var>x3</ruleml:var>
15     </swrlx:individualPropertyAtom>
16   </ruleml:_body>
17   <ruleml:_head>
18     <swrlx:individualPropertyAtom swrlx:property="hasUncle">
19       <ruleml:var>x1</ruleml:var>
20       <ruleml:var>x3</ruleml:var>
21     </swrlx:individualPropertyAtom>
22   </ruleml:_head>
23 </ruleml:imp>
```

Listing 2.2: Example SWRL rule in human readable and in XML syntax.

Modern reasoners (e.g., Pellet [63]) allow to augment OWL ontologies with SWRL rules to express conditions and relations that must hold between concepts of an ontology. We will employ SWRL rules in Sect. 7.4.1 to specify synchronization rules between domain models.

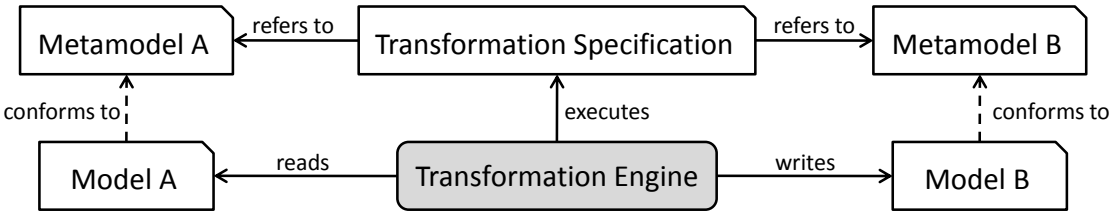


Figure 2.7: Schematic representation of a model transformation specification.

2.4 Model Transformations

A *model transformation* is the process of transforming one or more input models into a set of output models. Such a process entails a *transformation specification*, which defines how models are transformed and a *transformation engine*, which executes a given specification. The transformation engine takes a set of input models conforming to a set of input metamodels and creates a set of output models. The latter must in turn conform to a set of output metamodels. This relationship is depicted in Fig. 2.7 for the simple case where exactly one input and one output model exist. Transformation specifications consist of *transformation rules*.

From Fig. 2.7, one can also see that the transformation specification refers to the metamodels instead of concrete models. This definition on the metalevel allows to handle arbitrary input models as long as they conform to the same metamodel. In the course of this section one will also recognize that there are special transformations—so called *in place transformations*—where the input model is the output model.

Before looking at concrete transformation approaches (Sect. 2.4.2), as well as languages and systems (Sect. 2.4.3), a classification of existing model transformations will be conducted in Sect. 2.4.1. This way a structured picture of the various existing approaches can be obtained. For a general introduction to model transformations consider [64].

2.4.1 Classification

To classify model transformation approaches, Czarnecki and Helsen collected a set of criteria [65, 66] and formalized the relations between these criteria as feature trees. According to this classification, model transformation approaches differ in the employed transformation rules, the rule application (scoping, scheduling, directionality and strategy), their relationship between source and target models, as well as the extent to which they support traceability.

Czarnecki and Helsen do further divide these features. For example, transformation rules can be parameterized and use different kinds of patterns and pattern representations. Also, different kinds of typing can apply to rules. Model transformations can either create new target models or update existing ones. The latter can be performed by

	horizontal	vertical
endogenous	Refactoring	Formal refinement
exogenous	Language migration	Code generation

Table 2.1: Orthogonal dimensions of model transformations (from [67]).

extending the existing target model or also allow destructive changes. Transformation rules can be applied deterministically, non-deterministically or interactively.

For the scope of this thesis an important property of model transformations is directionality. According to Czarnecki and Helsen, rules can be either *unidirectional* or *bidirectional*. Unidirectional rules allow to transform models from a source domain to models of a target domain, but not the other way around. Bidirectional rules allow to transform in both directions. The latter can be further divided into bidirectional rules and rules consisting of complementary pairs. According this and the features listed above, numerous existing transformation approaches and tools were classified in [65].

The work on classifications for model transformations was continued by Mens and van Gorp in [67]. Here, the distinction between *horizontal* and *vertical* transformations was introduced. A horizontal transformation operates on the same level of abstraction, while a vertical one transforms models across abstraction levels. Language migration (e.g., translating from one programming language to another) is an example for the first category. Code generation is a representative of the latter.

Second, another orthogonal dimension was introduced—*endogenous* and *exogenous* transformations. For endogenous transformations, the input language is equal to the output language (i.e., both conform to the same metamodel). In contrast, exogenous transformations *translate* from one language to another, their input and output metamodels are different. To illustrate the orthogonality of the two dimensions, Table 2.1 lists examples for all four categories spanned by the two types per dimension. This categorization of model transformations into four distinct types is important, since the synchronization of models heavily depends on the kind of transformation that was used. Different transformations imply different treatment during synchronization.

Mens and van Gorp also proposed more classification criteria [67]. To name a few, they considered the level of automation, the complexity, the degree of preservation, correctness guarantees, composition and generality, bidirectionality, traceability and change propagation, usability, conciseness, performance, scalability, extensibility, interoperability and standardization of transformations. Again, traceability, bidirectionality and change propagation are the most interesting properties w.r.t. our work.

In summary, one can say that there are many criteria according to which model transformations can be characterized. However, not all of these properties are important in all contexts. Still, the classification criteria substantially help to set limits for the applicability of synchronization approaches to specific types of transformations.

2.4.2 Transformation Approaches

This section gives an overview of existing transformation approaches. These are not limited to model transformations in the literal sense, because transformations were used long before MDSD became prominent. Consequently, some transformation approaches are applied to models even though they were not originally designed to. To distinguish approaches, their underlying formal foundation must be investigated. For concrete transformations, this formal grounding is reflected by the rule specification mechanism and the associated semantics. As one will see shortly, there are also transformation approaches that have very little formal grounding and use informal semantics only. Nonetheless, they are listed here, because of their practical wide-spread usage.

An initial overview of model transformation approaches was presented in [68], but the list of covered approaches has not been exhaustive. The transformations according to the Common Warehouse Metamodel (CWM) [69] and Extensible Stylesheet Language Transformation (XSLT) [70] standards are rather languages than approaches. Thus, we will discuss them in Sect. 2.4.3. In addition to graph transformations and text-based transformations—being mentioned in [68]—we also consider ad-hoc transformations and term rewrite systems. The former are important, because they are often used in practice. A discussion of the latter is needed to understand the relation between operating on trees and graphs, which is the case for graph rewrite systems. Also, the problem of synchronizing artifacts when representing them as trees has been addressed in the literature and these approaches may help to develop solutions for graph-based artifacts.

Ad-hoc transformations

The most simple way to transform models, is to use an ad-hoc transformation. For example, one can employ a program written in a GPL to transform models. Such transformations are often imperative and not based on explicit transformation patterns. The expressiveness of a GPL does also imply that few assumptions can be made about the outcome of a transformation or its termination. Analyzing a transformation written in a GPL is hardly possible, which renders this approach the most inapplicable one if one cares about guarantees on the outcome of transformations.

Besides the expressiveness that limits the analysis, ad-hoc transformations are often performed on the level of concrete syntax. Instead of modifying logical elements of models their concrete syntactic representation is transformed. This is in contrast to the subsequent approaches which operate on well-defined structures (e.g., trees or graphs). Transforming on the level of concrete syntax is problematic for various reasons.

First, transformations can produce syntactically incorrect target models, because the transformation is not aware of the target metamodel. Second, transformations need to parse the concrete syntax to locate interesting elements of a model, which is error prone and should therefore not be part of the transformation process. Third, programs that

transform models, contain pieces of concrete model syntax, which makes them not only verbose and difficult to read, but also ties transformations to the concrete syntax. As these two concerns (i.e., representing a model with symbols and transforming a model) are independent of each other, they should not be mixed up.

A slightly more advanced way to transform models is to use programs that modify or translate models on the level of abstract syntax. Such programs are aware of the involved metamodels and do perform changes using provided model operations. Opposed to working on concrete syntax, such transformations are independent of concrete model representations and ensure the syntactic correctness of the transformation output. However, the fundamental problem of the limited analysis (i.e., no guarantees can be given about termination and correctness) is still present.

Having listed all these deficiencies exposed by ad-hoc transformations one must admit that many practitioners enjoy to use them. Speculation can be made about the reasons for this behavior. The effort needed to learn dedicated transformation languages or to familiarize with the formal foundation of a new transformation approach may be one reason here. The strong belief that one can solve any problem easily using a GPL may be another. Also, sophisticated tool machinery to transform models has not been available in the early days of MDSD. Thus, practitioners were forced to use alternative methods to transform models. In any case, ad-hoc transformation are used and approaches to model synchronization must take this into account.

Logic-based Transformation

Similar to using an imperative GPL one can also employ programming languages that are based on logic to implement transformations [68]. Examples include Prolog [71], Mercury [72] or F-Logic [73]. In contrast to object-oriented or procedural languages, logic-based languages are often declarative and provide means to specify patterns. Unification can then be used to match patterns in source models. Assuming one uses only a certain subset of the capabilities provided by logic languages, logic-based transformations can be analyzed to higher degree compared to transformations written in object-oriented or procedural languages.

Term Rewriting

Term rewriting is a formal computation model, which is based on the replacement of subterms according to a set of *rewriting rules*. A term representing the input data is reduced by applying rules that match subterms. The result of the computation is the term that is obtained if no rewrite rules are applicable anymore.

From a formal point of view, terms are trees and can therefore be used to represent tree-structured models. However, in this thesis, models are considered to be attributed, typed graphs conforming to the EMOF standard. Such graphs form a superset of all

tree-structured models. For some applications, this restriction may be acceptable, for others, term rewriting is not sufficient. Nonetheless, Term Rewrite Systems (TRS) can be analyzed w.r.t. important properties such as termination and confluence.

Term rewriting is used in program transformation, where it is called *tree rewriting* and has been implemented in various tools (e.g., TXL [74] or Stratego/XT [75]). It has also been shown that term rewrite rules can use the concrete syntax of the language that is transformed [76], which renders such transformation systems very useful in practice.

Graph Rewriting

Graphs provide a natural formalism to represent object-oriented models. Using nodes and edges, both model elements and their relations can be easily expressed. In addition, *type graphs* can be considered the correspondent to metamodels. They define different types of nodes and edges, which restrict the set of valid graphs that conform to a type graph. Another extension to graphs—attributes—are similar to properties of metaclasses and can therefore be used as their natural representation.

To transform graphs, Graph Rewriting Systems (GRS) [14] were developed as an extension of TRS. They have been successfully used in different application areas. General applications of GRS can be found in [77]. More specific applications to software engineering have also been studied [78]. This includes the specification of object behavior, the evolution of software architectures and the formal declaration of static and dynamic semantics. The diversity of applications shows how versatile and powerful GRS are.

GRS are also referred to as Graph Grammar (GG) and graph transformation. Rewriting graphs (as implied by the term GRS) and constructing them from an initial graph (as superimposed by the term GG), is sometimes considered to be different. But, one can also consider construction as a corner case where the initial host graph is an axiom.

According to [14], different classes of GRS can be distinguished using three criteria: a) the used notion of a graph, b) the condition under which a production may be applied, and c) the way the result of such an application is constructed. Examples for classes of GRS include Node Replacement GRS, (Hyper-)Edge Replacement GRS, Double Push-Out (DPO) systems, Single Push-Out (SPO) systems, attributed GRS, Triple Graph Grammar (TGG) [79] systems and High-Level Replacement Systems (HLRS). A particularly important class of GRS in the context of RTE are TGG systems as they support sophisticated bidirectional transformations.

Triple Graph Grammars Andy Schürr introduced the notion of TGG rules in [79]. These build a superset of pair grammars and allow the generation of different rewrite rules. For example, left-to-right transformations, right-to-left transformations or consistency analysis are supported. A TGG rule consists of three graphs (a left-hand, a right-hand and a correspondence graph). The latter collects information about the transformation history. Figure 2.8 shows the general schema for TGG rules.

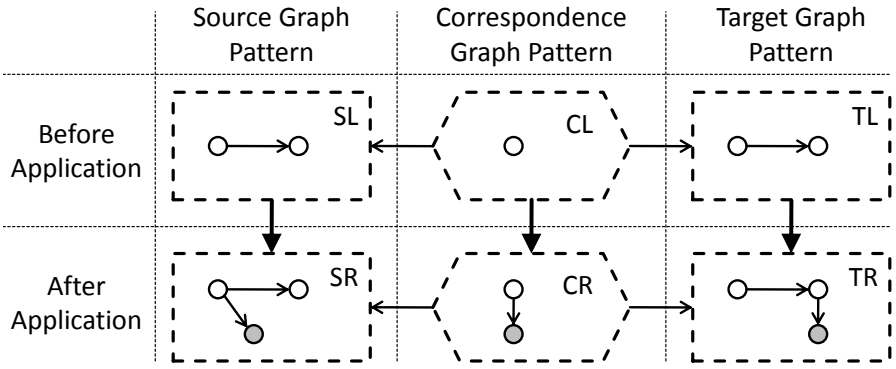


Figure 2.8: Schema of a TGG rule.

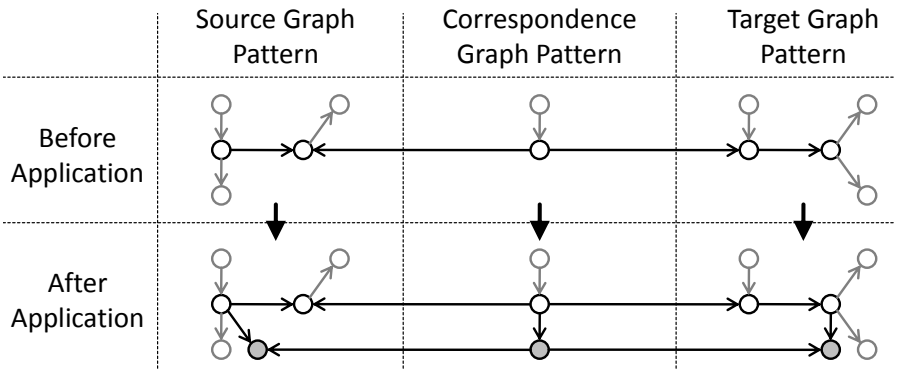


Figure 2.9: Embedding of a TGG rule.

For each of the three graphs a new version is given, which includes the changes implied by the application of the rule. Often these *before* and *after* views are merged into one graph where new or deleted parts of the graph are tagged. To apply a TGG rule to a concrete graph triple, a match for each of the graph patterns is searched. This match embeds the rule in the concrete graph and is then replaced by its corresponding *after* pattern. This is shown in Fig. 2.9.

Here, three concrete graphs are matched against the TGG rule shown in Fig. 2.8. To apply a TGG rule, the patterns specified by the left-hand side of a TGG rule must be matched (i.e., found) in the graph that is subject to transformation. Nodes that are not part of the match are depicted in gray shade. When the TGG rule is applied, the left-hand sides (shown in the upper half of Fig. 2.8 and 2.9) are replaced by the right-hand sides of the TGG rule. The new nodes and links are filled with gray color.

One can see that the general idea of TGG rules is similar to basic GRS. But, the graph that represents the left-hand and right-hand sides of TGG rules is separated into three

distinct parts as opposed to just one for basic GRS. This distinction allows to describe a synchronous evolution of three related graphs. As a consequence, TGG rules can be used to synchronize models, which is why they are important in the context of RTE.

Grunske et al. [80] proposed to use graphical TGG rules for the specification of model transformations. In contrast to textual specifications, the graphical representations are closer to the subjects of the transformations, if graphical models are transformed. However, it must be noted, that the graphical representation of TGG rules is only closer in terms of the used type of concrete syntax. Since TGG rules are specified at the level of abstract syntax, while models use concrete syntax, TGG rules do not reflect the natural appearance of the involved models. Furthermore, Grunske et al. showed how to translate graphical TGG rules to textual Tefkat specifications.

TGG rules were also utilized to transform between UML class diagrams and relational database schema models [81]. Using the TGG approach, a bidirectional transformation is established that can handle inheritance, non-persistent classes, as well as simple and complex attribute types. However, the presented TGG rules cannot handle the propagation of attributes to subclasses, because no notion of recursion is available.

Königs and Schürr extended the usage of TGG rules in [82] to a typical tool chain and showed how to integrate MOF-based metamodels. As in [81], TGG rules are determined to formalize the relations between different metamodel elements. These rules are then transformed to *operational graph rewriting rules* which can be used for synchronization or consistency checking. It is interesting to note, that the used TGG rules are monotonic, which means deletion of nodes or edges is not allowed. The reason for this is the arising graph parsing problem. Whether the absence of a deletion mechanism is a problem in real world scenarios is still an open question.

One major advantage of TGG rules, is the specification of rules that can be used for different purposes [82]. Other approaches specify distinct rules for consistency checking, forward and backward propagation of changes. When using TGG rules, only a single set of rules serves all objectives at one time. This allows for easier transformation specification and ensures that all (derived) rule sets implement the same semantics.

2.4.3 Transformation Languages and Systems

Based on the different transformation approaches outlined in Sect. 2.4.2, various transformation languages have been proposed. This section starts off with a classification of the most important ones. As enumerating all existing languages is out of the scope of this thesis, a selection is made. This selection is based on the relevance of the languages from a practical point of view and w.r.t. this thesis.

To give an overview of the transformation languages that are covered by this section, Table 2.2 summarizes some important language properties. It gives information about the type of the language (declarative or imperative), the kind of syntax used (textual or graphical) and the semantic foundation.

Transformation Language	Type	Syntax	Semantics
QVT	Hybrid	Hybrid	Informal
ATL	Hybrid	Textual	Informal
MISTRAL	Declarative	Textual	Informal
MOLA	Procedural	Graphical	Informal
GReAT	Declarative	Graphical	Graph Rewriting
Story Diagrams	Hybrid	Graphical	Graph Rewriting
GTXL	n/a	Textual	n/a
AGG	Declarative	Graphical	Graph Rewriting
PMT	Procedural	Textual	Informal
MoTMoT	Hybrid	Graphical	Graph Rewriting
Atom[3]	Declarative	Graphical	Graph Rewriting
VIATRA	Declarative	Graphical	Graph Rewriting
Tefkat	Declarative	Textual	Logic-based

Table 2.2: Classification of transformation languages.

QVT

Query View Transformation (QVT) [83] is a set of related transformation languages that was standardized by the OMG. It consists of a two level architecture. On the lower level, *QVT Core* can be used to perform pattern matching and to check conditions over variables (i.e., elements and attributes of models involved in the transformation). These basic operations can already be employed to transform models, but yield very verbose transformation specifications because of the simplicity of the operators. Based on this, *QVT Relations* can be used to specify relations between models declaratively. More complex patterns are supported on this higher level and trace information can automatically be gathered when executing transformations.

Transformations written in QVT Relations are translated QVT Core specifications. This is similar to the translation of high level programs to instructions of a virtual machine and also pictured as such in the QVT standard [83].

QVT is not formally based on the TGG formalism, but the relational part of QVT exposes many similarities. A comprehensive comparison between QVT and the TGG approach can be found in [84]. In the QVT standard itself, the semantics of the QVT languages is only specified informally. To obtain a more formal representation of the semantics of QVT, different approaches have been published [85, 86]. Unfortunately the QVT committee did not include these suggestions in new versions of the standard[3].

As the name QVT Relations indicates, the aim of this language is to allow for bidirectional transformations. However, the QVT specification contains some inconsistencies

[3] as of October 2010

which can cause problems in this regard [87]. Nonetheless many industrial transformation engines adopt the QVT standard.

ATL

Atlas Transformation Language (ATL) [88] is a model transformation language and tool developed as part of the ATLAS Model Management Architecture (AMMA) [89] platform. ATL uses textual syntax to define transformation rules. It adopts parts of the QVT specification, but does not strictly implement the standard (cf. [90] for a comparison of ATL and QVT). ATL does also rely on OCL to express conditions and constraints. Thus, ATL is a hybrid language that provides both imperative, as well as declarative concepts. ATL transformation rules are unidirectional. Thus, ATL cannot be readily employed in RTE.

However, ATL is very popular among practitioners, which can probably be explained by the fact that ATL was one of the first stable and mature transformation tools in the Eclipse ecosystem. Also, ATL is built on top of Eclipse Modeling Framework (EMF) [28], which is heavily used in the industry. The relevance that ATL has gained in the modeling community does amplify the need for RTE systems that can deal with unidirectional transformations. Rewriting all existing transformations using a bidirectional language is most probably not an option for practitioners.

Technically, ATL comes with a compiler that translates transformation specifications to a lower level representation, which can be executed by the ATL virtual machine. Also, quite recently attempts have been made to specify the semantics of ATL formally [91]. ATL does not provide built-in support for traceability, but transformations can be extended to emulate such functionality [92].

MISTRAL

Another language that can be used to transform models is Multiple IntenSion TRAnsformation Language (MISTRAL) [93]. MISTRAL is based on the observation that depending on the metalevel(s) a transformation is defined and execute on, different model transformation scenarios emerge. The authors explore reasons why there is no uniform language to handle all these scenarios. They point out the different meanings of generalization and instantiation on the various metalevels as a root cause. As a consequence, they propose a framework where the transformation language is separated from these two concepts. This framework is called MISTRAL and relies on mappings from models on all levels (M3-M0) to a generic model, which models the `instanceOf` relation explicitly.

MOLA

MOdel transformation LAnguage (MOLA) [94] is a procedural, graphical model transformation language. Using graphical syntax loops, patterns and statements can be ex-

pressed. The original paper on MOLA [94] showed two example transformations—a translation of class diagrams to relational models and the flattening of state charts.

The authors of MOLA, state that it was designed to be an *easily readable transformation language by combining traditional structured programming [...] with rules based on simple patterns* [94]. While readability is difficult to measure, a clear disadvantage of the approach is the complexity of the languages operators. Using constructs from programming languages (e.g., `while` loops), renders the analysis of transformation programs difficult. Using MOLA programs for bidirectional transformations does not seem feasible or is at least not considered in [94].

GReAT

Graph Rewriting and Transformation (GReAT) [95] is a transformation language that is based on graph rewriting. To specify source and target models, UML class diagrams are utilized. Elements of an additional (third) domain model define crosslinks between the source and the target. The underlying graphs are typed and attributed multi-graphs. Rules in GReAT can contain elements that are explicitly tagged with `new` (or `delete`) to indicate that they have to be created (or deleted) while executing the transformation. The authors of GReAT examine their approach using three scenarios: 1) a transformation of hierarchical concurrent state machines to finite state machines, 2) a Matlab Simulink Stateflow to Hybrid Automata conversion, and 3) a preprocessor for Java to add aspect code. The approach does not take bidirectional transformations into consideration.

Story Diagrams

Story Diagrams [96] are the transformation language that is used in the Fujaba [97] tool suite. Based on graph rewriting, graphical rules—so called patterns—can be specified. These patterns describe the transformation of object graphs. Elements in Story Diagrams can be tagged as `new` or `required`, which indicates whether elements must be present to apply a certain pattern or whether they must be created once a pattern matches. Story Diagrams do also allow for control structures (e.g., conditions and loops).

To execute a Story Diagram on a concrete object graph (or model), Fujaba translates the diagrams to executable Java code, which in turn modifies the input according to the diagrams contents. Recently, interpreters for Story Diagrams have also become available [98]. Story Diagrams have been successfully applied in various application areas, for example for Round-Trip Engineering [99], tool integration [97] and real-time system development [100].

The formal semantics of Story Diagrams was originally defined by a translation to Progres [101] productions [96]. But, the translation of sorted, ordered and qualified associations to Progres was still missing. Consequently a number of semantic issues were detected and outlined in [102].

GTXL

Graph Transformation Exchange Language (GTXL) [103, 104] is an exchange format for graph transformation rules based on XML. GTXL documents contain sets of rules together with positive and negative application conditions. GTXL can be used to exchange rules for different kinds of transformation techniques. Different techniques are tagged by their particular name. Thus, the semantics of the rules stored in a GTXL document is determined by the tools that process the rules. The GTXL format itself does not establish any restrictions in this regard. Thus, GTXL is purely considered with the specification of an abstract and concrete syntax for graph transformation rules. Semantics is not covered.

AGG

The Attributed Graph Grammar System (AGG) [105] is a tool developed at the TU Berlin. It is based on algebraic graph transformation [106]. Thus, AGG relies on a solid formal foundation. AGG supports various types of analysis that can be performed on transformation rules and input graphs. For example, it can parse graphs, check them for consistency and detect conflicts using critical pair analysis.

AGG uses an interpreter to execute transformation rules and can be flexibly used for different applications, because it supports to add attributes to graphs. Thus, arbitrary data can be represented by and transformed based on the graph formalism. AGG uses type graphs to model the set of graphs that can be transformed. Also, AGG supports negative application conditions to control the execution of rules on a fine-grained basis.

PMT

PMT [107] is a transformation language developed by Laurence Tratt, which is defined as a DSL in the *Converge* language [108]. PMT is a change-propagating transformation language. That is, instead of creating a new target model upon each execution of a transformation, the source and the target models are updated only. Changes made to one of the models are propagated to the opposite model. This implies an initialization phase, if the target model does not exist yet. In this case, PMT works like a unidirectional language. The target model is created by executing all changes required to build the source model from the scratch (i.e., assuming an empty source model first).

To specify how changes are propagated, PMT uses rules that consist of patterns, conditions and actions. When requested by the user, PMT propagates collected changes according to the specified rules. As PMT rules are Converge functions, the transformation is implemented by the execution of these functions. Since Converge is a full-fledged object-oriented programming language, the analysis of PMT rules is restricted.

MoTMoT

MoTMoT [109] is a model transformation tool that is based on Story Diagrams [96]. The original implementation of Story Diagrams in the Fujaba tool suite [97] was not based on common standards (e.g., it used a proprietary metamodeling language and serialization format). In contrast, MoTMoT employs UML and a specific profile for Story Diagrams. Thus, MoTMoT leverages the standardization of Story Diagram modeling. Among other things, MoTMoT has been used to apply refactorings to models [110]. Currently, there is activity in the Fujaba community to standardize both the syntax and semantics of Story Diagram models, which continues the goals of MoTMoT.

Atom^3

Atom^3 [111] is a metamodeling environment for visual languages. Besides its ability to specify visual languages and generate tools from these specifications, Atom^3 provides support to execute transformations between different languages. The different types of transformations supported by Atom^3 can be found in [112]. The formal basis for Atom^3 is graph transformation.

VIATRA

The VIsual Automated model TRAnsformations (VIATRA) [113] framework is a set of tools to visually define transformations for models that are based on the UML and the XML Metadata Interchange (XMI) [114] standard. VIATRA can, for example, be used to define static semantics in a declarative and pattern-based manner. Also, the dynamic semantics of models can be specified using graph transformation rules.

Tefkat

Tefkat [115] is an open-source model transformation engine based on EMF. Tefkat implements a declarative, textual variant of QVT. Transformation rules in Tefkat consist of patterns that have syntax similar to Structured Query Language (SQL) (e.g., `FROM` and `WHERE` clauses). Tefkat rules do support reflection to access metamodels during the execution of transformations. Tefkat also entails a debugger for transformation rules.

Due to its implementation, which is based on logics, Tefkat requires transformation rules to be stratified. For example, rules cannot specify a pattern that requires an object to not exist and then create this object if the rules matches. Since Tefkat is not based on a dedicated logic engine, but implemented from scratch, it is difficult to say whether this restriction is bound to the implementation or the approach in general. For example, the are subclasses of graph transformation that can stratify rules automatically [116]. Unfortunately, Tefkat seems to be inactive since 2007[4].

[4] cf. `http://tefkat.sourceforge.net/`

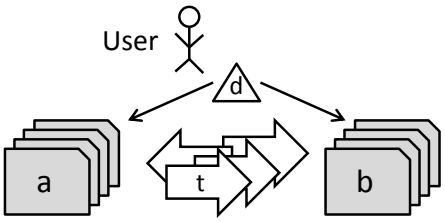


Figure 2.10: Schematic overview of a generic synchronization scenario.

2.5 Model Synchronization and Round-trip Engineering

Our approaches to RTE that will be presented in Chap. 6, 7 and 8 share the goal to automatize the synchronization of software artifacts. To give a high-level introduction to model synchronization and consistency management, we will recapitulate the general properties of such scenarios in this section.

First, there are artifacts (e.g., source code and documents) that shall be kept in sync. These can capture arbitrary kinds of data. The only requirement that is placed upon them in the context of this thesis, is their formal definition by a metamodel. That is, artifacts must be represented as models. Second, one or more transformations establish a relation between artifacts. Usually artifacts are derived from each other, but other types of transformations (e.g., multi-directional ones) are equally valid. This situation is depicted in Fig. 2.10.

The involved artifacts are denoted by letters such as a and b, whereas the transformations between them are called t. There do not need to be multiple transformations, a single one is sufficient to create a model synchronization scenario. Also, in Fig. 2.10 a single transformation step is depicted only. If one faces a scenario with multiple transformation stages, each stage needs to be dealt with individually.

The need for model synchronization is caused whenever *global consistency* is required. In conformance with [117, 118], we define this term as follows:

Definition 1: ***Global consistency*** *is a property of a set of models that is defined by a set of boolean constraints. If a set of models meets these constraints, it is considered to be globally consistent. If not, the models are considered to be inconsistent. The set of global consistency constraints induces a global consistency relation that holds for all consistent sets of models. In contrast to global consistency,* ***local consistency*** *is defined upon a single model and does therefore not refer to multiple models.*

We define consistency as a boolean property. Therefore we can use a relation for its formalization. But, there are other authors who argue that models can be consistent to a certain degree only [5, 119], because inconsistencies cannot be or should not be

completely avoided. In such situations, consistency should be defined as a percentage, but for the scope of this thesis, we do not consider this generalization.

To synchronize multiple related models, it is vital to be able to decide whether artifacts are consistent w.r.t. each other or not [120]. Whenever a user applies changes (d) to one of the involved artifacts, synchronization is required. If transformations are unidirectional (cf. Sect. 2.4.1), changes made to a model that serves as a source model (i.e., as input for a transformation) are easy to handle. The respective transformation can simply be executed once again, yielding a correct target model. If there are transformations for both directions, as in Fig. 2.10, changes made to target models can also be handled the same way by executing the respective transformations. However, this is rarely the case, and even if transformations for both directions are available one must make sure that they are inverses of each other.

The general, more difficult situation is present, if there are unidirectional transformations only. When a user applies a change to a target model, the question is, how all related models, particularly the involved source models, need to be changed to restore global consistency. More specifically, the question how to obtain consistent states automatically is a main subject of RTE. Therefore, we define RTE informally as follows:

Definition 2: ***Round-Trip Engineering*** *is the branch of software engineering that entails the design and application of techniques and tools to automatize the process of establishing global consistency across related software artifacts.*

While preserving consistency is a primary goal of RTE, a second goal—the *conservation* of artifacts—is almost equally important. For example, there can be arbitrary many source models that yield the same target model w.r.t. a given transformation. However, we might want a source model that is close to the existing one, instead of a randomly chosen one from the set of valid source models. We want to restore global consistency, but also conserve the original source model.

In other cases, there might not be a source model, that yields the desired output w.r.t. the transformation at all. Thus, both the source and the target models need to be adjusted to restore global consistency. Again, we would like to obtain a set of models that is close to the ones before the change. Therefore, we define *conservation* as:

Definition 3: ***Conservation*** *is the degree to which relevant properties of a set of models is preserved by the application of a set of changes.*

Of course, for practical applications the notion of *relevant properties* requires a formal definition to automatically decide whether changes are preserved or not and consequently how close models are w.r.t. the relevant properties.

One can see that global consistency and conservation are dual. Changes applied by users require RTE techniques both to reestablish a consistent state and to conserve relevant properties of the involved models. Consistency is a property of the state of models, whereas conservation is a property of the changes applied to models.

2.6 Traceability

The term traceability is defined in [121] as follows:

> The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.

A *trace* can therefore be considered as a single link between two related artifacts or elements thereof. In the context of MDSD, traceability has also been defined as *the runtime footprint of transformation execution* [66].

In general, traceability amounts to the quantity and quality of traces established during the development of a software system. A high degree of traceability is established if the relations among different artifacts are very well known. On the contrary, traceability is only accomplished to a very limited degree, if few relations are known or none of them are known at all. Poorly developed systems do have no documented information about the dependencies between artifacts. For example, the connection between the system's implementation and the requirements that were recorded upfront is lost.

If requirements do not change during development and after deployment, traceability may not seem an important goal to achieve. Unfortunately, few systems are static in this sense. Most do have changing requirements during development (e.g., because new knowledge about the system's domain is acquired) and after their initial start of operation (e.g., because the system's environment does change). Thus, traceability is an important aspect of software development, in particular when changes need to be applied to artifacts and the implications of these changes need to be known.

As traceability is concerned with relations among artifacts, one question is which kinds of relations exist. The corresponding literature reveals that there are various different types of relations. Starting with the very simple and generic trace type `traceTo`, one can add more types like `dependsOn`, `modifies`, `responsibleOf`, `rationaleOf`, `validatedBy`, `verifiedBy` and `assignedTo` [122]. Some authors even propose the usage of much more types of traces [123] or facet-based trace information [124].

The selection of trace types that are needed within a particular development process heavily depends on the application domain, the size of the application, its predicted lifetime and external constraints such as governmental obligations. It does also depend

on the kinds of analysis that are performed on software artifacts. If one wants to run a change impact analysis [125], trace links that allow such evaluation (e.g., `dependsOn` links) are required.

This specificity posed on the trace information is reflected by the variety of metamodels that have been proposed for modeling trace information over the last years [123, 126, 127]. These show that no single model for traceability information will suffice all needs. Rather, configuration and extension mechanisms must be employed to provide flexible and adaptive means to trace relations across artifacts.

Once appropriate trace information is available, a variety of applications can use this information. The well known change impact analysis is only one application for traceability [128]. Trace information can also foster the comprehension of systems, enable a coverage analysis (e.g., to compute which requirements are covered by an implementation), help to find unused artifacts (i.e., to conduct an orphan analysis), be employed while debugging transformations and assist model synchronization approaches [66, 129]. The last application area is particularly important in the scope of this thesis.

Trace information can be either acquired automatically or provided manually. For example, trace links can be created automatically while executing model transformations [124]. If artifacts are derived manually (e.g., when creating use case diagrams according to a requirements document), traces must be established by hand. Even though the second option is tedious, it can be inevitable for some applications.

In the scope of this thesis, trace links are used to capture the relations between derived (i.e., obtained by a transformation) and copied (i.e., obtained by a composition) artifacts. We will see that these kinds of trace links can store the relation between redundant pieces of data and how it can be used for synchronization purposes. In particular the RTE approach that is based on backpropagation and presented in Chap. 6 heavily relies on trace information.

3
Problem Analysis

The need to synchronize software artifacts to enable the integrated usage of multiple heterogeneous tools has been present since the early days of software engineering. The term RTE, which encloses this problem area, has been identified as a key enabler for MDSD [120]. Consequently, various approaches have been proposed to provide solutions for model synchronization or model-based tool integration (cf. Chap. 10). While many approaches target specific RTE applications and most of them provide a reasonable solution w.r.t. their objectives, the overall support for RTE is still in its infancy.

To determine the reasons for this discrepancy between the large number of approaches targeting RTE and the limited availability of generic solutions, a more extensive analysis of the problem is needed. As it will turn out over the course of this chapter, there is multiple, orthogonal aspects of the problem as a whole. Each section in this chapter will elaborate on one of these aspects.

We will start with a general analysis of the reasons that cause redundancy in software development. These are shown as part of Fig. 3.1 and will be examined in Sect. 3.1. A particular set of reasons is inherently connected to MDSD, which is why we will conduct a separate, more close inspection of these in Sect. 3.2.

In general, there are two possibilities to address problems that are implied by redundancy found in software artifacts. First, one can try to *avoid redundancy*. Second, one can *employ synchronization techniques* to deal with unavoidable redundancy. Both approaches are complementary and they expose their own sets of problems. These are analyzed in the next sections. Section 3.3 particularly presents the blocking factors when trying to avoid redundancy during software development. In Sect. 3.4 we discuss problems that are related to the synchronization of artifacts.

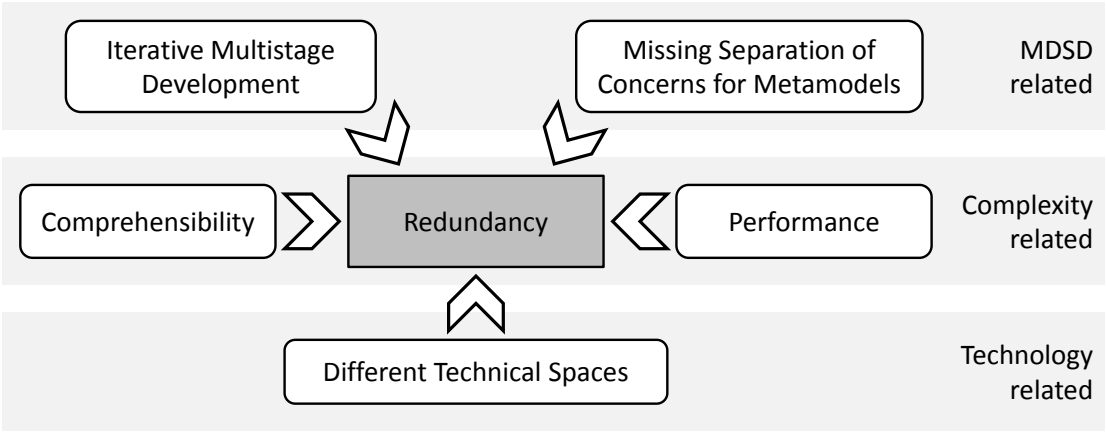


Figure 3.1: Causes for introducing redundancy.

After analyzing problems of RTE in detail, we will draw conclusions in Sect. 3.5 and identify the most relevant parts that must be addressed to obtain more generic solutions to RTE. Thus, the analysis performed in this chapter is the basis for our conceptual framework in Chap. 4 and the concrete approaches to RTE proposed in Chap. 6, 7 and 8. Chapter 5 builds the foundation required to apply these and other approaches to RTE in different technical environments. Thus, it does not present a particular solution approach to RTE, but rather enables more widespread applicability or RTE technology.

3.1 General Causes for Redundancy in Software Development

In this section, we will briefly present some general causes for redundancy in software artifacts that are not particularly bound to MDSD, but can potentially be observed in any development process. The following is a (incomplete) list of scenarios where duplicate information is introduced for different reasons.

- Limitations of single tools

  A very simple but common reason for the introduction of redundancies are technical limitations of tools. For example, a programming language without support for aspect-orientation can enforce developers to duplicate and scatter code. Consequently, multiple instances of one and the same fact exists. Since the choice of tools is often restricted by external parameters, the resulting redundancies can hardly be avoided.

- Limited tool and data integration

Software development processes involve a variety of tools. The number of these tools can be very small, but also scale up to several dozens. Ideally, all these tools should understand one language thus communicating to each other via a unified file format or database. In reality, different tools handle different input formats. For example, word processors or spreadsheet calculators that are used to collect requirements store data that is also collected in issue tracking systems. But, both use completely different means to store data (e.g., files vs. databases). To integrate such tools one must map the corresponding formats to each other, thereby replicating the description of the software.

- Performance limitations

  The structure and layout of a system specification can be different depending on its purpose. High level specifications are used for analysis and verification, while low or medium level descriptions can be executed or optimized. Depending on the complexity of the specific task and the overall system specification, representations may need to be condensed or extracted to ease performing a particular development step. For example, a call graph analyzer extracts the control flow from a program and stores it in a call graph before running the actual analysis. Again, information is duplicated. This practice is intended and reasonable, but also implies an increased amount of redundancy.

- Comprehensibility boundaries

  Software artifacts can grow large and may therefore be difficult to understand or process. Extracting specific (smaller) parts from a (huge) artifact is a well known technique to conquer complexity. Creating such views on artifacts introduces redundancy because facts are replicated from the original artifact. This is similar to the previous item, but the client that demands a partial view on the system specification is the developer, instead of a particular tool.

Besides these very general reasons for the introduction of redundancies, there are more specific causes which are particularly related to MDSD. We will elaborate on these in the next section.

3.2 Redundancy in MDSD

3.2.1 Duplication in Iterative Multistage Software Development

For decades the majority of research in software engineering was focused on *forward engineering*. Development processes were specialized to quickly implement systems, often without considering the tremendous costs of maintaining and evolving these systems in the future. Today we are faced with a multitude of existing systems which are not

only subject to maintenance, but also need to be extended. Thus, the existing implementations often need to be reverse engineered [130]. In parallel, new development methodologies (e.g., MDSD and Model-Driven Architecture (MDA) [131]) have emerged, which do explicitly connect artifacts that are created in different stages of software development. Both the ignorance of the importance of Reverse Engineering and the new multistage paradigms heavily increase the need for synchronizing software artifacts. This section will shed more light on the implications induced by this change of mind.

MDSD puts emphasis on chains of models. One starts with abstract descriptions of a system and performs refinements yielding more concrete models and finally executable code. This procedure exposes similarities to the linear nature that can be observed in early development process models. Similar to the processes, the steps (i.e., the creation and refinement of models) are considered to be executed sequentially—one after another. However, this is not feasible for the same reasons processes were redesigned to support iterating in cycles and returning to previous steps as needed. The closer a system gets to its concrete implementation, the more likely design faults are detected. Once a new system is actually used by people (or test personnel), change requests are filed and parts of the design and the implementation may need to be modified.

In both cases (i.e., returning to an earlier development stage and going back to change a more abstract model) the effort spent previously should not be lost. A system's implementation should not be completely redone when the design is changed partially. Refinements made to models must be kept even if more abstract models are modified. In other words, the artifacts created in all stages of a development process need to be *synchronized*. When running a linear process, the synchronization issue does not arise, because people can stop caring about artifacts created earlier. But, if these early artifacts (e.g., abstract models) are used for any other purpose, they must be in sync with any derived and potentially changed documents.

For example, if the UML model of a system is used to generate the implementation for a new platform, it must be synchronized with the code that is actually running on existing platforms. On the contrary, if the UML model is not used at all after code generation, synchronization is not needed.

To summarize, one can ascertain that the historical forward orientation of software development methodologies yielded linear sequences of artifacts. Once a subsequent stage of development was reached, the documents of the previous one were obsolete. Duplication of data was not problematic and synchronization among artifacts was therefore not a main concern. However, the shift toward iterative, cyclic and multistage development changes this. Now, artifacts of all development stages are kept and duplication needs to be dealt with. Artifacts must be kept in sync across development stages and abstraction levels. Redundancy cannot be resolved by discarding artifacts anymore. What has been ignored or at best handled manually in the past grows to a new magnitude within the context of evolving systems, iterative processes and MDSD.

3.2.2 Absence of Concern Separation in Metamodeling

In [132] the term *Separation of Concerns* was first mentioned. Edsger W. Dijkstra wrote this essay to emphasize the importance of studying different aspects of complex problems in isolation rather than looking at all of them at once. If one mixes various issues, the result will be confusion rather than gaining valuable insights about the matter of subject.

Creating software systems is a complex problem. Thus, separating concerns while designing and implementing such systems is good advice. However, for different reasons this advice is often not taken seriously (e.g., by mistake) or cannot be followed easily (e.g., because of present technical restrictions). Sometimes, developers may basically not be aware of the fact that they are mixing up concerns. They might try to get things done quickly without worrying about the consequences. At other occasions, they might be well aware of the fact that they are mixing concerns, but technical constraints force them to do so. For example, in the absence of an aspect-oriented language a cross-cutting concern may yield duplicated information in a system description.

If one does violate the Separation of Concerns principle—for one reason or the other—redundancy arises. If artifacts that contribute to the specification or implementation of a software system address the same concern, facts are duplicated and changing one artifact causes the need to change another. If, on the contrary, every artifact targets exactly one concern, which is not targeted by another, system information is much more isolated and less duplication can be observed. There are less dependencies across artifacts if they address separate concerns and consequently there is less redundant data.

Paradoxically, in the context of MDSD separating concerns if often not supported by the employed languages and tools. Common object-oriented metamodeling languages (e.g., EMOF) do not provide built-in support for concern separation. Rather, distinct patterns must be used to implement metamodel composition [97]. These patterns allow to create, extend and thereby reuse metamodels, but there is no first-class support for a strict separation of elements that belong to a specific concern. One cannot easily split metamodels into different concerns, establishing a logical grouping of elements. As a consequence, there is no support for merging sets of concerns. Composition of different orthogonal parts of a metamodel similar to the use of hyperspaces [133] is not available in common, widespread languages. This shortcoming has been identified and addressed by recent publications [134].

If systems were fully designed and implemented in terms of separating concerns, development artifacts would exhibit much less redundancy, fewer dependencies and relations that one must take care of. Or to put it the other way round, the extent to which a system description violates the principle of concern separation accounts for the amount of synchronization needed [135]. While there is good reason to strive for perfect separation when developing software, some mixed concerns—at best only the ones present for technical reasons—will always remain.

3.3 Anticipation for Future Tool Integration

We have already argued in Sect. 3.2.1 that the majority of software development processes employed in the past was forward-oriented. Few effort was put into considering the costs of making significant changes at later development steps like testing or maintenance. A similar observation can be made w.r.t. the anticipation of future RTE scenarios.

Existing software systems are rarely built in a way that allows for later integration with other systems. For example, enterprise applications can achieve integration by accessing the same databases. While this is a feasible approach if new applications are added, it poses many problems when integrating existing applications which are already bound to their custom database schemata. Then, depending on the design of the involved applications, heavy changes can be necessary to achieve integration. Similar situations can be observed for other kinds of applications and technical foundations as well. In any case, one can say that the less precautions have been taken to allow later integration, the more effort is needed to achieve interoperability. A great amount of duplication found in software development is caused by such late integrations.

This insight raises the question, why tool interoperability is usually dealt with at integration time only. As RTE is also concerned with the integration of tools we could probably avoid a huge amount of redundancy by anticipating RTE scenarios in the first place. Since it seems obvious that taking the right precautions significantly eases this task, it is reasonable to ask why this is not been done.

First, one can argue that the huge variety of future integration scenarios cannot be foreseen. This is certainly true, since one cannot anticipate every potential tool integration scenario. Thus, minimal effort is put into designing for interoperability, because developers are convinced that implementing unknown requirements is not meaningful. Second, interoperability is usually realized by dedicated interfaces. This reflects the awareness for the fact that certain parts of an application may need to be replaced or reconnected differently in the future. However, the anticipation for integration is still limited. Only parts which are encapsulated behind appropriate interfaces can be reconnected. One can denote this kind of anticipation as foreseen integration.

If systems could be developed in a way that also unforeseen integration scenarios can be supported, the effort to integrate tools and to exchange their data could be substantially reduced. However, this requires first-class support for data and behavioral integration.

3.4 Employing Synchronization to Handle Redundancy

To deal with any amount of redundancy that cannot be avoided for one of the reasons mentioned earlier, synchronization techniques are required. While there is quite a number of these available, certain new problems can be observed here. The subsequent subsections analyze issues that are connected to the synchronization of redundant data.

3.4.1 Domain Restrictions of Existing Approaches

When analyzing the existing body of work in the context of RTE, quite a number of publications can be found. While few papers address the problem of RTE as a whole by giving definitions, goals and requirements [120, 136, 137], most publications deal with concrete synchronization problems. For example, the database community has spent effort on investigating when and how relational views can be updated [138, 139, 140, 141]. The modeling community has also created a variety of techniques to synchronize models [83, 87, 142, 143]. In particular, the synchronization between models and generated code has been studied [99, 144, 145, 146]. Moreover, approaches that are specific to a particular formalism (e.g., string data [147, 148]) or domains (e.g., framework modeling [149]) have been presented.

Given this amount of existing work, one could assume that building new RTE systems is fairly easy. Unfortunately, the contrary can be observed. Often no decent actions are taken to deal with redundancy at all. Synchronization is performed manually yielding the usual problems of increased effort and error-proneness. Even if some kind of automation is in place, it is often based on ad-hoc implementations, neglecting existing generic solutions. The important question arising here, is why the adoption of available approaches is not done at least to a reasonable extent.

From our point of view, the main blocking factor for the adoption of all these existing methods is the limited possibility to transfer them to other domains. For example, the view update problem, which the database community investigates on, is tightly bound to the relational calculus. Data structures which rely on other formalisms can hardly be treated the same way. The same argument applies to the work that was performed to synchronize string data. While the results are very convincing, they cannot be easily applied to graph structures as we find them in MDSD.

The work on model synchronization using bidirectional model transformations and RTE for template languages has been examined in isolation although both problems are very similar in their nature. If only one stops treating textual artifacts conceptually different from models, both problems collapse to a single one and more generic solutions are rendered possible.

Moreover, the ontology community has spent considerable effort on aligning and matching formally represented knowledge [150]. The results of this work start to diffuse into other communities, but have not been taken in to the necessary extent yet.

These observations lead to the conclusion that there are multiple barriers that prevent the adoption and thereby the easy implementation of new RTE systems. There are technical issues, because each community maintains its own specific tools and techniques, which cannot easily be transferred. Also, there are conceptual problems, which are caused by the different underlying formalisms, which are employed by particular RTE approaches. We address these two barriers in Chap. 5.

3.4.2 Complexity of Synchronization Specifications

Given a situation where data is replicated, the need to perform synchronization arises and a respective specification is required. Such a specification formally defines how to propagate changes between data and all its replications. It can either state relations between data elements and therefore have a declarative nature or consist of concrete actions which must be taken when changes are made to replicated data. The latter is said to have an imperative style.

While there are many tools, which can in principle be used to synchronize data—actually any (model) transformation tool can serve this purpose—few are really suitable for this task. This is caused by a number of special requirements in the context of RTE.

First, synchronization requires to propagate changes in multiple directions. Modifications made to the original data need to be visible in all replicated versions, and likewise altering a copied piece of data must trigger changes in the original. Thus, the application of unidirectional specification mechanisms is problematic if the opposite transformation cannot be derived automatically. Moreover, there are cases where no transformation for the opposite direction exists. For example, if there are multiple, equally valid choices for propagating a change, a decision must be made by the user. If this decision cannot be automated, no bidirectional transformation can exist.

Second, synchronization rules are often defined at the level of the abstract syntax of the involved artifacts. While this allows tools to process rules, engineers have a hard time writing these specifications. This is caused by the fact that using abstract syntax yields more verbose rules. To illustrate this, consider a situation where one needs to create an Abstract Syntax Tree (AST) for a program instead of writing the respective source code. Besides the verbosity of abstract syntax, developers are more familiar with the concrete syntax as they deal with it more often.

Third, many tools favor graphical or declarative synchronization rules. When executing these rules, it is difficult to find errors because debugging is more complex for this kind of rules. Graphical languages use a combination of graphical and textual elements. Often the textual elements are hidden (e.g., constraints are edited in separate views and not the graphical canvas itself). Therefore, it is difficult to follow the execution (i.e., the application of rules). While the rule interpreter proceeds it can highlight the graphical elements which are currently evaluated, but the hidden, textual parts of the rules cannot be shown. In contrast to textual rules where one can easily follow the evaluation of conditions and actions, because all rule elements are textual, this is a drawback.

For declarative synchronization specifications a similar problem can be observed. Since the steps that must be executed to evaluate a declarative rule are derived (i.e., declarative rules cannot be executed as they are), there is no one-to-one correspondence between the steps performed during synchronization and the elements of a rule. While declarative specifications must certainly be preferred over imperative techniques, this gap between what is specified and what is executed, poses a problem for users.

Problem area	Negative Implications
Synchronization requires bidirectionality	Rules are required for each direction, or bidirectional rules must be used, which do not always exist
Specifications use abstract syntax	Rules are difficult to read and write
Specifications use graphical syntax and/or declarative style	Rules are difficult to debug
Specifications are based on syntax (not on semantics)	Rules cannot be reused automatically

Table 3.1: Synchronization specification problems.

Fourth, synchronization is specified w.r.t. the (abstract) syntax of the artifacts. Rules refer to elements of metamodels. This renders transformations to work on a purely syntactic level. The meaning of the elements in the metamodel is not taken into consideration. For example, synchronizing two different artifacts which contain information about persons requires to explicitly state the equality of the concept `Person` in one artifact, which may be called `User` in the other one. If one could process the meaning of the two concepts formally, this synchronization rule can be derived automatically.

Table 3.1 summarizes the problems mentioned before. One can see that some problems conflict with each other. For example, graphical and declarative specifications certainly ease their creation and are particularly useful for bidirectional transformations, but they are difficult to debug. Similarly, basing rules on syntax eases the creation and understanding of rules, but hinders the their reuse.

Chapters 6 and 7 will partially address the problems identified in this section. For example, the backpropagation-based approach to models synchronization is applicable if no bidirectional transformation can be found. The ontology-based synchronization is based on the formal semantics of artifacts rather than their abstract syntax. Other approaches which do also target some of these problem can be found in [151] and [152].

3.4.3 Simultaneous Treatment of Synchronization Dimensions

Redundancy can manifest itself in two different ways. First, the same data can be represented differently. For example, information about users of a software system can be stored using the XML format, plain text or a comma separated file. While all three representations may capture the same information, synchronization is required when one of the artifacts changes. Because of the fact that the information contained in all formats is based on exactly the same concepts, but rather persisted using different representations, we call this kind of synchronization *technical synchronization*.

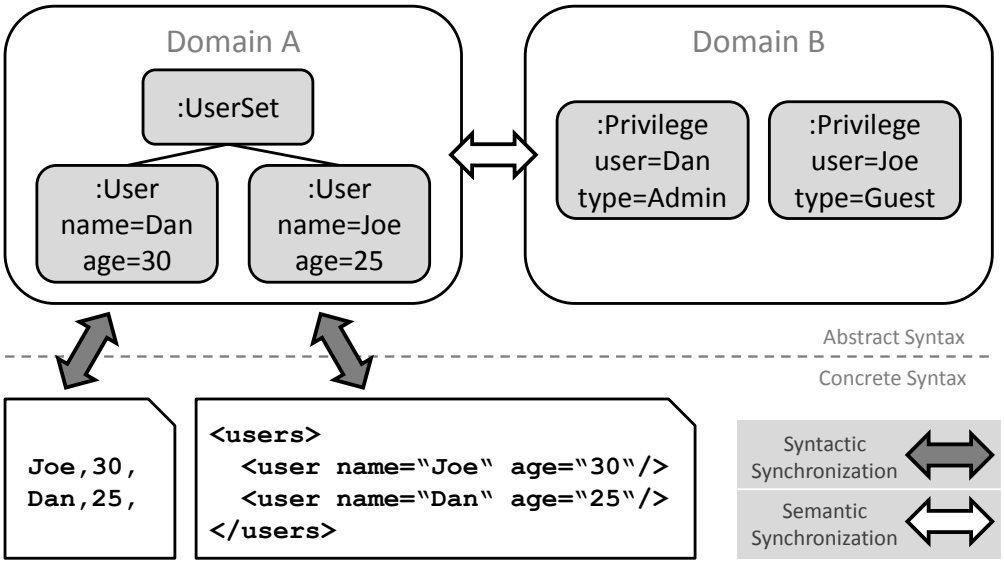


Figure 3.2: Syntactic vs. semantic synchronization.

One can find different representations of a piece of information if different *concrete syntax* is used for the same *abstract syntax* (cf. Sect. 2.1.3). Also, there are multiple technologies to specify abstract syntax, which allows to encode the structure of information in different technical ways. For example, one can use an XML Schema Definition (XSD) [153] document or a context-free grammar to specify the same type of information.

In contrast, some synchronization scenarios are more complex because they involve artifacts that share only a subset of their information. In this case, the synchronization is not only concerned with the transformation of data to other technical representations, but involves a semantic mapping between artifacts. This mapping must specify which parts of one artifact relate to parts in another one.

In Fig. 3.2, an example, which illustrates the difference between abstract and concrete syntax is shown. Starting from the lower left part one can see how different concrete syntax (e.g., comma separated values or XML) can be used to represent the same data. In this case the actual relevant data is depicted in the upper left corner of Fig. 3.2. The synchronization between this abstract representation and the concrete ones is therefore a technical one. On the contrary, the synchronization between the user domain (A) and the security domain (B) is a semantic mapping, because both domains do not use the same concepts. They do share some information (e.g., the user names), but they are not representations of exactly the same data. Rather one could say, that both domains partially refer to common objects of the modeled world.

As one will recognize in later sections, the two synchronization types are orthogonal. Consequently, they need different solution approaches and yield different tools to tackle

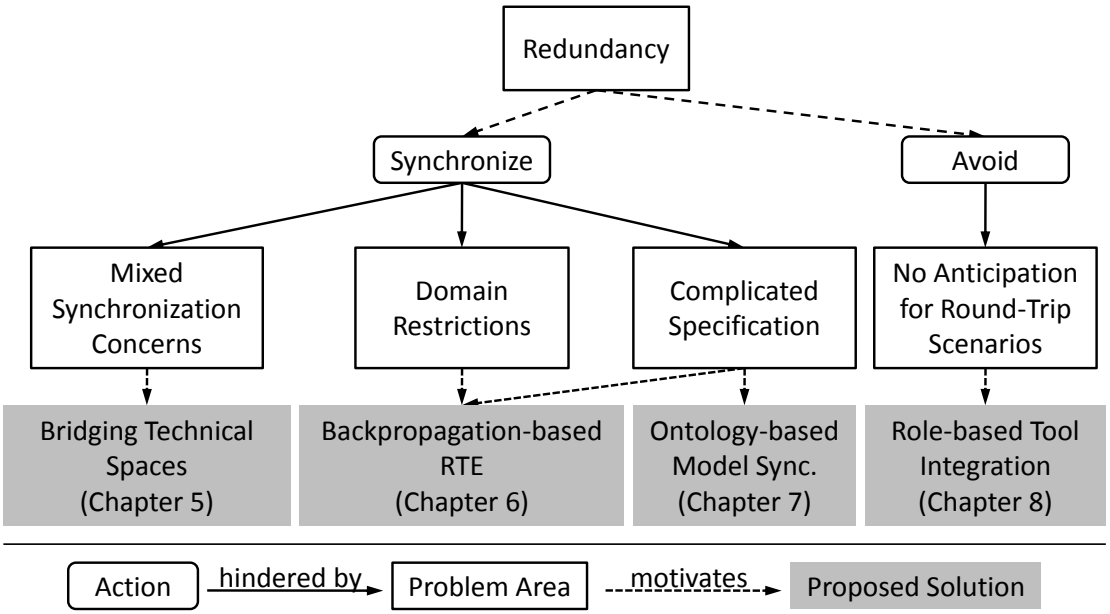


Figure 3.3: Identified problem areas and proposed solutions.

such synchronization scenarios in practice. Often, both synchronization tasks need to be performed sequentially. First, artifacts must be transformed to a common technical representation after which they can be synchronized semantically. One may also conceive the technical transformation as bridging technical spaces and the semantic one as establishing a conceptual mapping between two or more domains.

3.5 Conclusion

In this chapter we discussed various aspects that contribute to the fact that no widely applicable approaches to RTE are available. We have seen that the history of software development, which is dominated by forward-oriented methods, is one reason that many tools lack support for RTE. Also, the absence of standardized metamodeling facilities accounts for a great amount of redundancy we need to deal with. Moreover, even the metamodeling languages that are available today, do offer limited support for composition only. This renders clean separation of concerns difficult. Similar problems can be observed when looking at model transformations, where there is no strict separation between domain knowledge and technical detail. Again, redundancy is the result.

To avoid problems related to redundancy, there is basically two options. First, one can deal with redundancy by employing synchronization techniques. Second, one can try to reduce or even eliminate redundancy. Both options are shown at the top of Fig. 3.3.

Depending on which option one picks, further problems can be observed. For example, if the first choice is picked, the mixing of orthogonal synchronization concerns, the restricted applicability of existing synchronization approaches, and the complicated specification of synchronization rules need to be dealt with. These three problem areas are also shown in Fig. 3.3—in the bottom left part.

Each of the solutions for RTE proposed in this thesis targets one or more of the identified problem areas. Backpropagation-based RTE (Chap. 6) is a widely applicable synchronization technique, which can be used even in cases, where bidirectional synchronization rules do not exist. Ontology-based RTE (Chap. 7) does also try to ease the creation of synchronization specifications. Role-based RTE (Chap. 8) aims at eliminating redundancies and can serve as a method to anticipate future RTE scenarios. In addition, bridging technical spaces, which is discussed in Chap. 5, emphasizes on separating semantic and technical concerns when synchronizing artifacts.

Our conceptual framework (Chap. 4) defines the common grounding for all these approaches. It delivers common terms and provides a generalization of the concrete procedures that will be presented.

4

A Conceptual Framework for Round-Trip Engineering

The problem analysis conducted in the previous chapter revealed that certain problems in the context of designing RTE systems frequently reoccur. For example, we have seen that mixing different orthogonal synchronization concerns, as well as the lack of anticipation for future situations where RTE is required, can render the design process complex. These are very general problem areas that apply to many RTE applications.

Similarly, the review of related work in Chap. 10 discovers commonalities between different, often very specific RTE applications. Among other things, many approaches employ traceability information, rely on change propagation or provide means to reduce redundancies to a minimum.

To exploit the aspects that are shared across different RTE systems, we propose a unified conceptual framework for RTE. The goal of this framework is to deliver both terms and definitions that are required for the design of RTE systems (Sect. 4.1), but also to provide systematic advice on how to advance during this design process (Sect. 4.3). The primary means of our framework are partitioning and composition of software artifacts, which will be explained in Sect. 4.2.

Main contributions of this chapter are

C4-1 a terminology for RTE,

C4-2 a generic framework for RTE based on model partitioning, and

C4-3 a design process for RTE systems.

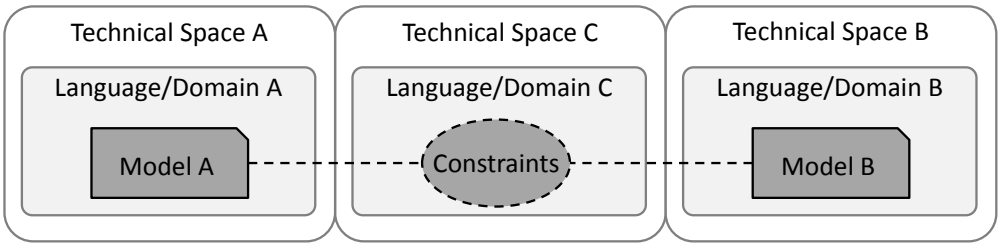


Figure 4.1: Conceptual view of a Round-Trip Engineering scenario.

4.1 Terms and Definitions

The goal of RTE is to keep a set of related artifacts synchronized. Whenever a change is applied to one of the involved artifacts, others might need to be adjusted to restore a consistent state. The notion of *consistency* can be expressed in terms of a set of constraints (cf. Def. 1). Also, artifacts that are subject to synchronization, can reside in different technical spaces [154]. This situation is depicted in Fig. 4.1.

Here, we can find three technical spaces that entail one language each. We will also use the term *domain* to denote such a language. Each language defines a set of valid artifacts using the concepts provided by the technical space. We will see examples for this in Chap. 5. For the time being it is sufficient to state that artifacts conform to a language and that languages reside in technical spaces. Also, the constraints that specify global consistency are expressed in terms of some language. Thus, these are located in a technical space too. In conformance with Fig. 4.1 we define a *RTE scenario* as follows:

Definition 4: *A **RTE scenario** is tuple $(t_a, t_b, t_c, l_a, l_b, l_c)$ where t_a, t_b, t_c are technical spaces and l_a, l_b, l_c are languages defined in these spaces. The languages l_a and l_b are used to specify models that must be synchronized, while l_c is used to express the constraints that must hold to consider models to be consistent.*

The three technical spaces do not need to be disjoint. If spaces overlap, domains can overlap too. Also, multiple languages can be used to express consistency constraints. These can even reside in different technical spaces. To denote systems that manage consistency in a RTE scenario, we define the term *RTE system* as follows:

Definition 5: *A **RTE system** connects the languages in a RTE scenario such that changes that are applied to artifacts conforming to l_a or l_b are processed and global consistency is restored.*

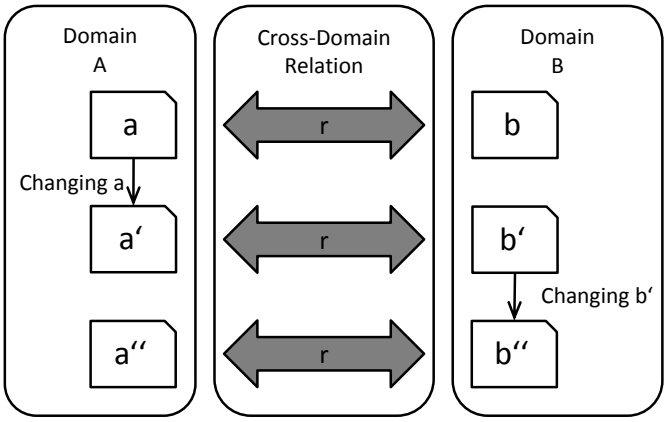


Figure 4.2: Artifact evolution in Round-Trip Engineering.

According to Def. 4 and 5, RTE systems are required to process artifacts from all involved technical spaces. Thus, the design of such a system incorporates both an understanding of the technical properties, as well as the logical relations between involved languages. In the following, we will first consider the latter concern, assuming all artifacts reside in the same technical space. Details on how to transfer artifacts to such a common space will be discussed in Chap. 5.

The evolution of artifacts in a RTE scenario and the adjustment of related artifacts to restore global consistency is shown in Fig. 4.2. Here, two domains (A and B) are depicted that contain one evolving artifact each (a and b). One can also find a relation between artifacts—the *cross-domain relation*. This relation is induced by the set of consistency constraints and needs to be restored whenever artifacts change.

In MDSD, the cross-domain relation is usually not defined arbitrarily, but established by a model transformation—a function that derives one artifact from another (cf. Sect. 2.4). As functions introduce a particular direction w.r.t. the domains shown in Fig. 4.2, the domain of the function (i.e., its input) is denoted as *source domain*, whereas its range (i.e., the output) is referred to as *target domain*. Using functions here, is more restrictive than the general case, where arbitrary relations are allowed. Defining transformations as functions implies that there is at most one artifact in the target domain for each source artifact. This rules out transformations that are non-deterministic. However, from a practical point of view, using functions only is fine.

To further analyze the relations between artifacts, consider Fig. 4.3. In the lower half, different kinds of transformations rules according to the classification of Czarnecki and Helsen [65] (Sect. 2.8) are depicted. We can find *bidirectional rules*, *invertible rules* and *non-invertible rules*. The first type can be directly used for both transformation directions. The second type cannot be employed directly to perform backward transfor-

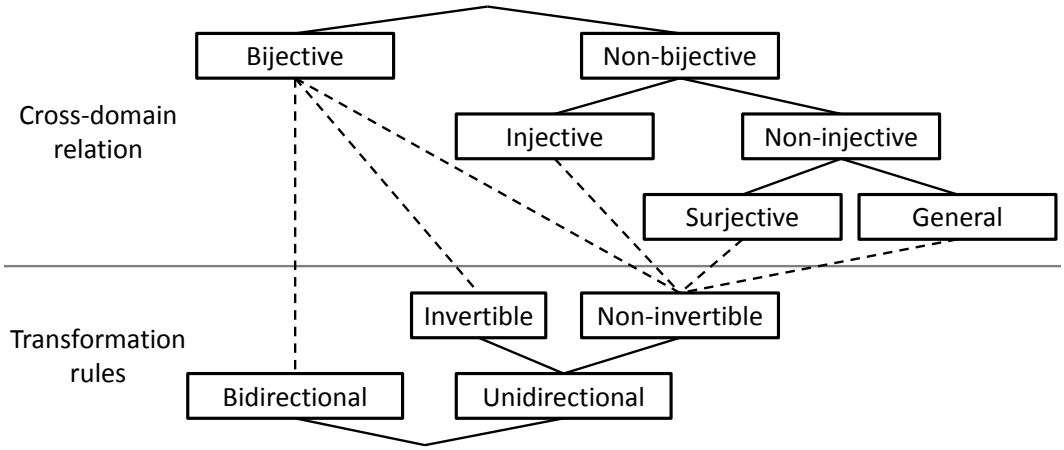


Figure 4.3: Classification of cross-domain relations and transformation rules.

mations, but allows to compute respective rules. After this computation, invertible rules can be used similarly to bidirectional ones. The third type is not invertible, in the sense that even though inverses might exist, they cannot be computed automatically.

In the upper part of Fig. 4.3, we have added different kinds of cross-domain relations. Depending on the concrete RTE scenario different such relations can be observed. We have classified these into *bijective relations*, where one source model corresponds to exactly one target model, and *non-bijective relations*. The latter are further distinguished into *injective*, *surjective* and *general relations*. Even though all bijective relations are both injective and surjective by definition, we refer only to those relations that have one property, but not the other. General relations are neither injective nor surjective. Thus, all leaf nodes in the upper tree denote mutually disjoint sets of relations.

When looking at the different types of transformation rules, one can observe that bidirectional and invertible rules create bijective relations between domains only. In contrast, non-invertible rules can create arbitrary relations. The former follows directly from the fact that a function f is bijective if and only if its inverse relation f^{-1} is a function. Since we assume that all model transformation are functions, bidirectional and invertible rules form such pairs (f, f^{-1}). To avoid confusion, we must add that this statement does only hold for rules that solely use either the source or the target model as input. Approaches that take previous artifact versions into account (e.g., Pierce's Lenses [155]) can of course create non-bijective cross-domain relations.

The relations that hold between artifacts are of utter importance in the context of RTE. As we will show later on, their properties determine the applicability of different approaches. Therefore, we will examine them in more detail—on the level of model elements instead of whole models. But, before doing so, we need to have a closer look at the information that is shared between two domains.

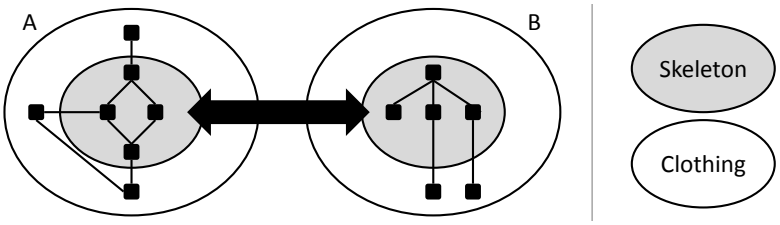


Figure 4.4: Partitioning models into skeleton and clothing.

In the most general case, artifacts that are related in a RTE scenario share some portion of their data. Whenever changes are made to this shared information, the corresponding artifacts need to be adjusted. In [156], this part of a model is denoted as *relevant model*. For reasons that will become obvious shortly, we use the term *skeleton* instead of *relevant model*, as depicted in Fig. 4.4.

Here, two models (A and B) are partitioned into two parts each. Each model is separated into its skeleton and its clothing. The latter is local to a domain and is not directly involved when synchronizing models. We call this part of a model *clothing* to emphasize the duality with the skeleton metaphor. As the shared information is captured by the skeletons only, synchronization is only performed between them.

Typical unidirectional model transformations use whole models (i.e., both skeleton and clothing), but produce only a skeleton in the target domain. The clothing in the target domain is usually created by developers (e.g., by filling implementation gaps or augmenting models with additional information). The fact that this clothing is added to the physical result of the unidirectional transformation, causes difficulties during synchronization. One cannot easily recreate target models, because the clothing gets lost if no precautionary actions are taken.

Formally, we can define a skeleton as follows:

Definition 6: *A **skeleton** is a model fragment that contains all parts, which are relevant w.r.t. a particular RTE scenario.*

To illustrate concrete examples for skeletons and clothings, consider Table 4.1. This table lists the types of elements that belong to the skeleton or the clothing. It is not complete, because the involved artifacts can contain many more types, but one gets an intuition about the skeleton concept.

In Table 4.1, we can also observe, that the skeleton depends on the concrete RTE scenario. For example, comments may not considered to be part of the skeleton when synchronizing UML class diagrams and Java code, but in the context of reconciling Java code and Hypertext Markup Language (HTML) documentation generated by JavaDoc, these comments are indeed part of the skeleton. Skeletons are scenario-specific.

Domain A		Domain B	
Clothing	Skeleton	Skeleton	Clothing
Database schemata		UML class diagrams	
Schema, Index	Table, Column, Foreign key	Class, Attribute, Reference	Package
UML class diagrams		Java code	
	Package, Class, Attribute, Reference, Method	Package, Class, Field, Field, Method	Statement, Comment
Java code		JavaDoc	
Statement	Class, Class comment, Method, Method comment	Class section, Class comment section, Method section, Method comment section	HTML

Table 4.1: Examples for skeletons and clothings.

To compute the part of a model that is relevant w.r.t. a specific transformation, Hettel et al. introduced the function *strip* [156]. This function extracts a relevant model given a full model and a transformation as input. Our concept of a skeleton is similar, but independent of concrete transformations. Thus, we define the function *cut* that derives the skeleton of a model as:

Definition 7: *A **skeleton selection function** $cut_X : X \to X$ maps a model from domain X to a model fragment of the same domain, removing all parts that are irrelevant w.r.t. a particular RTE scenario. It can be used to compute skeletons for arbitrary models of domain X.*

The function cut_X can also be considered as a filter function on models that selects exactly the parts of a model that are shared with another domain. We will discuss more details about the design of cut_X in Sect. 4.2, but for the time being, we can say that cut_X determines the model fragment that is preserved w.r.t. the cross-domain relation. The notion of *being preserved* is crucial in the context of RTE, because the relation between skeletons (i.e., the degree to which skeletons are preserved) determines the complexity of a respective RTE system.

Having said that cut_X is a filter function on models, we can make another observation. The examples listed in Table 4.1 select skeletons based on the type of model elements.

Our evaluation (cf. Sect. 9.1) confirms this observation. Even though we do not generally assume that *type-based skeleton selection* is applicable in all RTE scenarios, its frequent occurrence is worthwhile to mention.

Besides selecting skeletons based on types (i.e., depending of the concepts defined by the language of an artifact), one can also partition models based on their concrete contents. For example, model elements can be explicitly selected to be included in the skeleton, while all other elements are left out. We call this *instance-based skeleton selection*. One can say that this kind of partitioning is performed on metalevel `M1`, while type-based partitioning refers to `M2`. Moreover, the language specification concepts (i.e., the concepts to define languages) can form a basis for model partitioning. Thus, we define the following types of skeleton selection functions:

Definition 8: *Skeleton selection functions can either be* ***instance-based***, ***type-based*** *or* ***metatype-based*** *depending on the metalayer they depend on. Instance-based functions refer to metalayer* `M1`*, type-based functions refer to* `M2` *and metatype-based functions are defined on* `M3`*.*

Instance-based skeleton selection functions are used less frequently, because they cannot be reused for new models. Examples for type-based partitioning can be found in Table 4.1. A metatype-based partitioning will be employed by the role-based approach to tool integration in Chap. 8.

Assuming we have two functions cut_A and cut_B (i.e., one for domain A and one for domain B) two skeletons s_A and s_B can be computed. Since we define a skeleton as the part of a model that is shared across domains, there must be a relation between the two skeletons. Respectively, we call this a *skeleton relation*.

Now, our hypothesis is the following:

H1 If there are two functions cut_A and cut_B and an injective relation between the respective skeletons, one can build a deterministic RTE system.

Here, the term *deterministic RTE system* refers to systems where changes can be propagated to related artifacts in a unique way, whereas *indeterministic RTE systems* may involve the necessity to choose from a set possible propagation options.

4.2 Aspect-based Partitioning of Artifacts

According to our definitions in the previous section, separating models into parts that are relevant w.r.t. a particular RTE scenario—the skeletons—and parts that are not—the clothing—is an essential concept in our framework. Obviously, we do need to *partition* models, but also do we require means to *compose* them in order to obtain artifacts that contain both shared and local information.

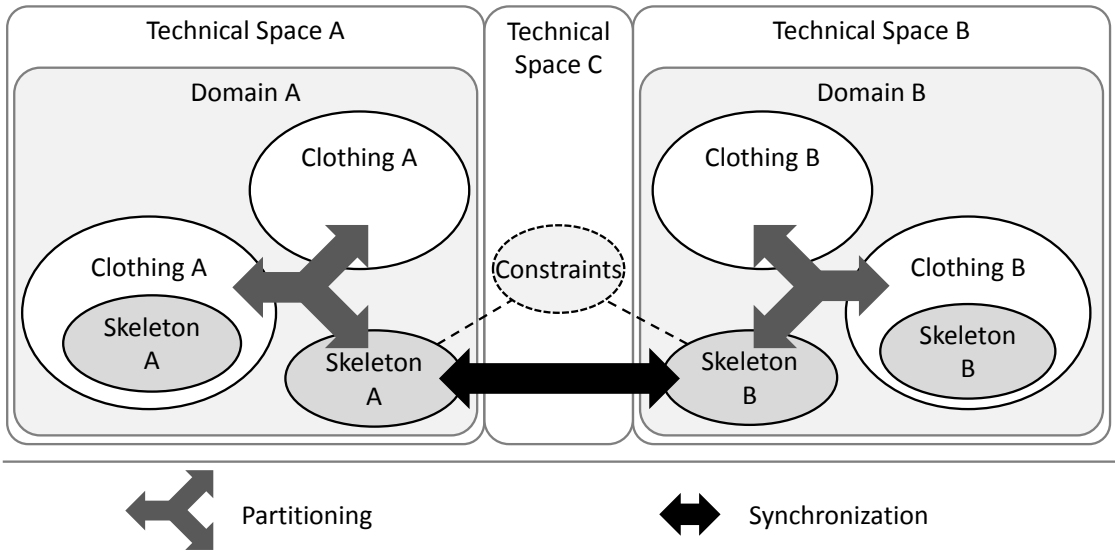


Figure 4.5: Partitioning, composition and synchronization of models.

The literature exposes various methods to compose and partition object-oriented software. For example, one can employ facets [157], roles [10], composition filters [158], hyperslices [133], mixin layers [159], or invasive software composition [160]. In addition to these methods, the logical partitioning of software descriptions —to distinguish parts that are correlated with different concerns—has been proposed. For example, a separation into *essence*, *administration* and *infrastructure* is presented in [161]. Here, parts that are essential to an application (e.g., data structures) are separated from administrative concerns (e.g., constraint checking and error handling). Also, parts that deal with communication and connection are split off.

We argue that building RTE systems is substantially easier if one of these techniques is employed to partition artifacts into skeleton and clothing. The general idea of this approach is depicted in Fig. 4.5. Later on, we will see concrete examples for partitioning and composition in Chap. 9.

For each domain involved in a RTE scenario, a partitioning cut_X is required that separates skeleton and clothing (depicted as gray arrows in Fig. 4.5). Once we have two such partitioning functions, the skeletons can be subject to synchronization (the black arrow in Fig. 4.5). The general goal of the partitioning is to obtain skeletons that are either isomorph, or that expose at least an injective relation between their nodes.

The technical implementation of cut_X can base on any of the techniques mentioned above. In our work we will frequently rely on Invasive Software Composition (ISC) [160] to perform this task as it provides flexible means to partition and compose models. In general, other composition techniques are most probably equally feasible here.

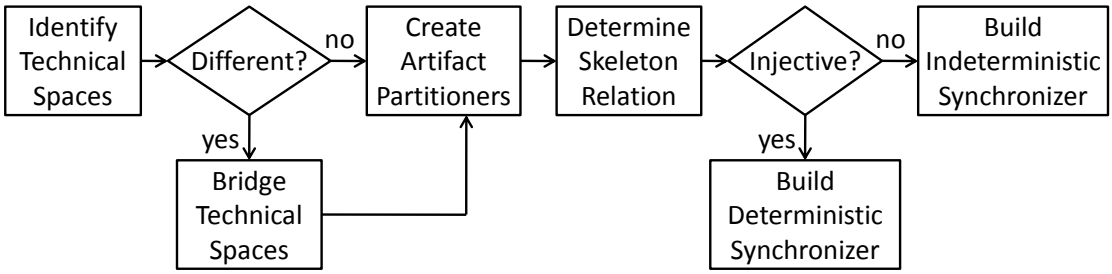


Figure 4.6: Generic design process for RTE systems.

4.3 Design Process and Round-Trip Engineering Patterns

Now that we have presented partitioning of artifacts, being a main building block of our framework, we can sketch a process that embodies the necessary activities to establish RTE support for concrete scenarios. Here, a concern that was already mentioned in Sect. 4.1, is the distribution of artifacts across different technical spaces. We propose to strictly separate all activities that transfer artifacts between such spaces from other activities that are concerned with logical synchronization. We have observed that mixing these two tasks yields to unnecessary complexity w.r.t. RTE. Other authors have held the same view in the context of language and tool integration [162, 163]. Thus, we consider bridging the different technical spaces involved in a RTE scenario as a prerequisite to build a respective RTE system.

In Fig. 4.6, a respective design process, that puts building bridges upfront, is shown. In Chap. 5 we present concrete examples for such bridges. Briefly put, these bridges allow to access and process artifacts that are created using different technologies in a uniform way. Once all artifacts reside in the same space, an analysis of the shared data can be performed. Based on the result of this analysis, we propose to partition models into skeleton and clothing. The former must enclose all data that is shared across artifacts and must therefore be involved in the synchronization. The latter entails data that is local to a particular domain.

Depending on the concrete relation that holds between skeletons, one must either build a deterministic synchronizer or an indeterministic one. This choice is not up to the designer, but rather dictated by the involved transformation (or constraints, respectively). In Chap. 6 and 8 we present concrete approaches that employ model partitioning.

The process depicted in Fig. 4.6 applies to elementary RTE scenarios—scenarios that include two domains only. If one needs to deal with more than two domains, sequences can be built from the basic pattern shown in Fig. 4.5. An example for a basic sequence that involves three domains (A, B and C) is depicted in Fig. 4.7. One can see that the artifact residing in domain B is partitioned in two ways. For the synchronization with domain A a skeleton S_{B1} is used, while the synchronization toward domain C is based

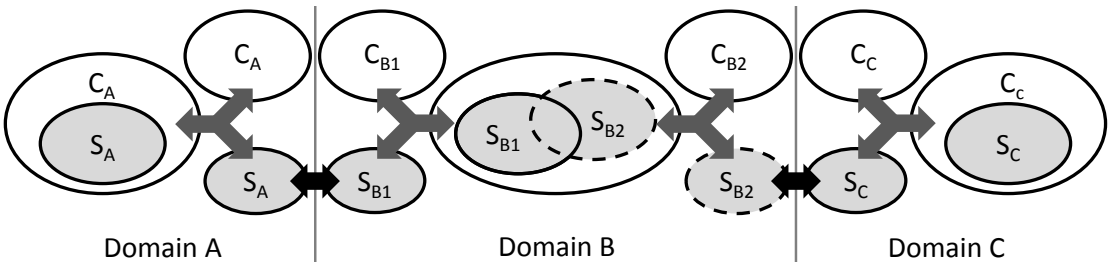


Figure 4.7: Composing RTE systems.

on skeleton S_{B2}. In Fig. 4.7, the skeletons overlap, but this is not a requirement. They can also be disjoint, as well as equal.

In addition to the simple sequence that can be built from two RTE scenarios, other typical RTE patterns can be formulated using the skeleton metaphor. For example, having multiple views on a global integrated model, corresponds to a multiple view domains $V1, \ldots, Vn$ which are synchronized with the domain G of the global model. In this case, the clothings $C_{V1}, \ldots, C_{Vn}$ are usually empty, because views shall be completely reflected in the global model. The skeletons for the views $S_{V1}, \ldots, S_{Vn}$ differ, which accounts for the fact the each view focuses on a different aspect of the global model.

Partitioning artifacts, as we propose it, is specific to single RTE scenarios. Consequently, specifications and implementations of skeleton selection functions cannot be reused across different RTE scenarios unless the same data is shared between two domains. However, the bridges for technical spaces are independent of concrete RTE scenarios and can therefore be employed in multiple scenarios.

5
Bridging Technical Spaces

When looking at the documents involved in software development processes, one can observe that artifacts are stored in heterogeneous formats. Source code is stored in plain text files, configuration data often resides in XML files, test data can be found in relational databases, images and documentation may use proprietary binary formats. All these formats are handled by specialized tools, which can read, process and persist the required information. In addition, any kind of artifact is accompanied by certain knowledge and particular skills to modify or create such artifacts. The entirety of these technical properties and the associated knowledge is called a *technological space* [164]. A similar, but more restrictive definition can be found in [154], where the term *technical space* is introduced (cf. Sect. 5.1 for a more extensive comparison of the two).

When artifacts shall be kept in sync, the first obstacle encountered is their different representation and structure. These differences are often present if artifacts reside in different technical spaces. They can also be observed, if artifacts residing in the same technical space, but use different concrete syntax. In any case, to synchronize related data, one must be able to transfer all information to a common representation. Transferring information involves both a *conceptual mapping* and a *technical integration*. The former is needed to relate the concepts found in different spaces (e.g., how to map tables to tree structures). The latter is needed to actually retrieve and persist information during synchronization. Both tasks will be subject to further explanation in Sect. 5.1.

Bridging technical spaces can be regarded as a prerequisite for *logical synchronization*. It converts the information in a RTE scenario to a common technical representation, which is an abstraction of the concrete artifacts containing the information. Bridging technical spaces, as defined above, is often merely conceived a technical issue rather than

a substantial research problem. However, the mapping between different technical spaces is far from trivial. The experience gained during the course of this thesis suggests that the ratio between the two aspects of RTE depends on the concrete scenario tackled. In some cases, the complexity of transferring artifacts across boundaries of technical spaces outweighs the actual effort required to reconcile elements across artifacts logically. In other cases, bridging technical spaces is rather easy (e.g., if they are identical or ready-made bridges exist), but the semantic mapping between the elements is very complex.

In any case, both problems need to be solved in order to address real RTE scenarios. This chapter addresses the first issue (i.e., how to bridge technical spaces), whereas logical synchronization aspects are subject to the subsequent chapters. After a definition and a general analysis of the requirements to bridge technical spaces (Sect. 5.1), this chapter does recapitulate some existing bridges between widespread technical spaces (Sect. 5.2). We consider only technical spaces, which are particularly relevant for our work. That followed, the bridge between the technical space of context-free grammars and object-oriented models will be sketched in Sect. 5.3. In Sect. 5.4, the mapping between object-oriented models and ontologies will be discussed. Then, the relation between role models and object-oriented models will be subject to further investigation (Sect. 5.5).

All three bridges are relevant for this thesis for multiple reasons. First, each bridge widens the applicability of the presented RTE approaches. Basically, more kinds of artifacts can be handled, if there is a bridge available that connects the space where synchronization takes places and the space where artifacts reside. In particular, the bridge between context-free grammars and object-oriented models is of practical relevance in this regard, as it allows to incorporate any artifact that is structured according to such a grammar. Because of the long history of compiler technology, which makes heavy use of grammars, the majority of software artifacts can be treated by using this bridge.

Second, each bridge allows to perform the logical synchronization of data in a technical space of choice. Once one can freely transform between different spaces, synchronization algorithms can be executed in any space that fits the particular needs of an application best. In addition, the space can be chosen depending on the readily available mechanisms for synchronization. This is the motivation for building bridges to the ontological space (Sect. 5.4) and role modeling (Sect. 5.5). Both do not enable the integration of a large set of typical software artifacts as the context-free grammars do, but they have particular properties that were considered to ease RTE.

The technical spaces presented in this chapter do not strive for completeness. There exist many more and new ones will emerge in the future. We present a selection of spaces, which are heavily populated by artifacts that can be found in software development. However, if new kinds of artifacts, which reside in different technical spaces, need to be synchronized, the general procedures outlined in the next section must be applied. Until now, there is no known procedure that is capable of bridging technical spaces automatically. Providing such a procedure is outside of the scope of this thesis, but the general steps that must be taken to build such bridges will be discussed.

Main contributions of this chapter are

C5-1 an analysis of the mappings required to bridge technical spaces in general,

C5-2 detailed inspections of mappings that are employed in concrete bridges, and

C5-3 providing the means to apply subsequent RTE approaches more widely.

5.1 Technical Spaces - Definitions and Properties

The term *technological space* has been initially defined in [164] as follows:

> A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities.

In [154], Bézivin and Kurtev continued their investigations on representing different technologies to enable their interoperability. They replaced the term *technological space* with *technical space* and gave a slightly more restrictive definition, which is:

> A technical space is a model management framework accompanied by a set of tools that operate on the models definable within the framework.

In [154] it is stated that the term *technical space* has been introduced in [164]. However, in [164] the term *technological space* is used exclusively. Thus, there is no clear differentiation between the two terms. Within this thesis, we will use the term *technical space*, assuming the latter, more restrictive definition is subsumed by the former, less restrictive one.

Prominent examples for technological spaces are XML, context-free grammars, binary files, and databases. Each of these technological spaces is connected with specialized tools that can be used to process information that is available in the respective space. Skilled software developers are not only familiar with the tools, but also have background knowledge about the used technology. They know the internal structures of their artifacts, the limitations and drawbacks. The relations between these aspects, that characterize a technological space (i.e., the more general notion), are depicted in Fig. 5.1.

One can distinguish between formal properties of a technological space, such as languages specifications used to define valid data within the space, or the API used to access or query existing data. Other properties are informal and difficult to grasp by a specification. This set of properties includes the skills or knowledge required to solve problems in a specific space. However, we consider only the formal aspects to be covered by the term technical space (i.e., the more restrictive definition).

In [154] Bézivin and Kurtev argue that the integration of different technical spaces is required to overcome the limitations of one space by transferring artifacts to another. In practice, the artifacts that are subject to synchronization do populate different technical

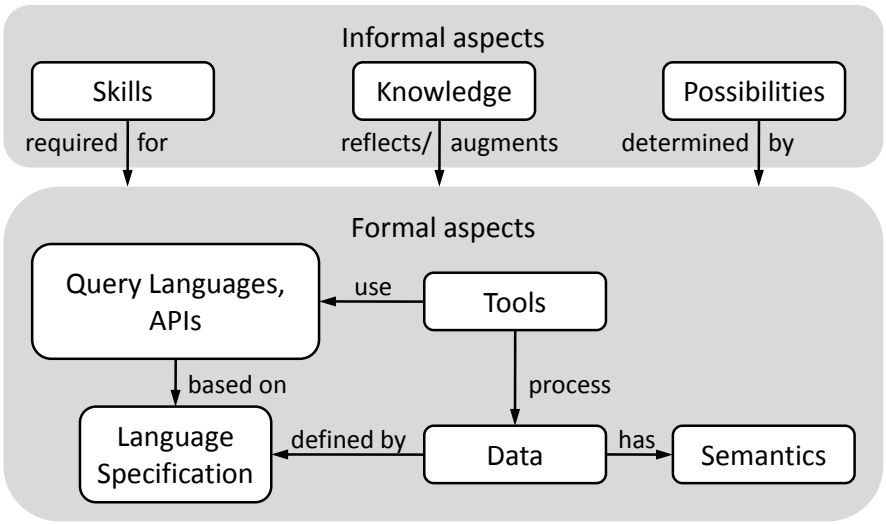


Figure 5.1: Aspects of technological spaces.

spaces. Thus, the integration discussion from [154] supports the claim of this thesis—bridging technical spaces as prerequisite for RTE—as it emphasizes the fact that artifacts reside in different spaces for good reasons.

A sketch for the construction of such bridges between technical spaces is depicted in Fig. 5.2. Here, we can find the main ingredients of technical spaces—language concepts, languages, artifacts and tools. The separation of technical spaces into three metalayers was introduced in [164].

It is important to note that bridging spaces is twofold. First, a *conceptual mapping* between the spaces is needed. This mapping relates the language concepts to each other (cf. [165]). It forms the basis and the frame for mappings between languages. Second, if concrete artifact types need to be mapped, the conceptual mapping needs to be instantiated by a *language mapping*. It is essential to recognize that two mappings (i.e., one on `M3` and one on `M2`) are required to transfer a particular type of artifact from one space to another.

The conceptual mapping connects two spaces as a whole. It maps between the concepts of the spaces and is independent of concrete artifacts or languages. For example, if information residing in the ontology space shall be synchronized with data present in the modeling space, ontological concepts (e.g., OWLClass) need to be mapped to modeling concepts. This conceptual mapping may, for example, state that classes in ontologies are mapped to classes in metamodels. It does not make any statement about mappings between concrete classes. This is different from mapping concepts of domain models, where artifacts reside in the same technical space, but share domain knowledge. We discuss specifics of these conceptual mappings in Sect. 5.1.1.

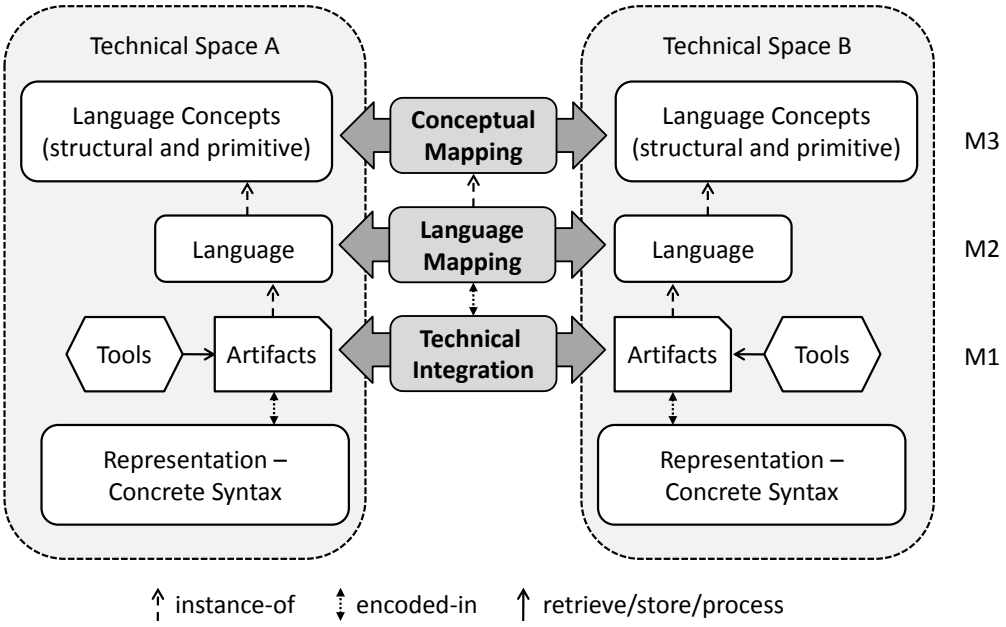


Figure 5.2: Mapping technical spaces.

Unfortunately, mapping concepts of technical spaces is not sufficient to synchronize data. Even in the presence of a valid mapping between ontological concepts and modeling structures, one needs to retrieve and store actual data that instantiates these concepts. We call this second mapping a language mapping as it is performed for two concrete languages. It couples two concrete languages, but not two entire spaces. The details of this second mapping will be discussed in Sect. 5.1.2.

The language mapping can be implemented in different ways. Depending on the kind of technical integration performed, either transformations or adapters are used. In [154], components that implement the language mapping between technical spaces are denoted as *projectors*. We will discuss both styles of integration in Sect. 5.1.3.

5.1.1 Conceptual Mappings

Every technical space has certain properties. First, information residing in such a space exposes some kind of structure. This structure follows certain rules, which are defined either formally or informally. Also, the information held in these structures is made of basic building blocks—primitive types. In addition, all valid structures have some meaning—they expose semantics. If we want to transfer information from one technical space to another, these aspects must be considered altogether. We call the entirety of this relationship between two spaces a *conceptual mapping*. The mapping is conceptual,

because it is not aware of the concrete types of data (i.e., languages). Rather, it establishes a relation between the concepts—both structurally and semantically—of two spaces. In this section, we will discuss the nature of such conceptual mappings more deeply, as well as the requirements that should be met by them in general.

Every piece of data that resides in a particular technical space adheres to the restrictions of this space. Typically, data is organized according some predefined structures. For example, object-oriented models use graphs as a structural paradigm. Other spaces (e.g., the XML space) use trees as an underlying formalism to arrange information. Structural concepts are typically used to construct complex units of information from smaller, less complex units. To establish a conceptual mapping between two technical spaces, the structural concepts (i.e., the means to construct complex data) must be mapped. Such a mapping must preserve important properties (e.g., the hierarchical order of information). Even though most technical spaces exhibit means to express complex data structures, this is not the case for all spaces. Some artifacts (e.g., binary files containing test data) do not make use of structure. Rather, these are organized in a flat manner. Thus, the amount of structure employed in a technical space can vary.

Besides the conceptual gap between data structures employed by different technical spaces, primitive data types must also be considered. Here, mapping data types of one space to types in the other is required. Conversion of data values may also be needed. For example, if one space offers a primitive type to represent dates, but another one does not, a reasonable translation is needed. For example, dates may be represented as strings using some canonical format. For technical spaces that make use of common basic data types such as integers, strings and booleans, mapping primitive types is quite simple. It does not involve any conversion, if the ranges of all primitive types are equal.

While mapping complex and primitive data types, one must always keep the semantics of the elements that are subject to the mapping in mind. For example, if tables (i.e., a structural concept of one space) must have unique names, the corresponding concept (e.g., XML complex types) should have a matching semantics. As the conceptual mapping is established on the level of the language concepts, only the semantics of the concepts is relevant here. The meaning of concrete languages does differ from language to language and must therefore be considered individually.

In general, semantics can be divided into its static and its dynamic aspects. While the former entails the properties that can be checked before execution, the latter describes the dynamic behavior of a software system, specified by an artifact. Thus, on the conceptual level, where we map the language concepts to each other, the dominating aspect is the static one. Dynamic semantics differs for concrete languages and must therefore be considered during the creation of the language mapping.

Depending on the concrete technical spaces to be mapped, there can be few semantic restrictions (e.g., for context-free grammars) or quite many constraints that must be taken care of (e.g., for ontologies). Unfortunately, few technical spaces have a formal definition for the static and dynamic semantics of their concepts. Even concrete

languages within one space are rarely accompanied by a formal definition of their semantics. Languages in some spaces do at least capture the static aspects (e.g., using attribute grammars in combination with context-free grammars), but formal descriptions of dynamic semantics are rare.

As a consequence, the correctness of a mapping (both between spaces and concrete languages) is usually difficult to establish. Sometimes there might not even exist a mapping which preserves all semantic properties when transferring data to a different space. As long as the properties that are relevant to the concrete application of a bridge between two spaces are preserved, this is not problematic. But again, this requires the formalization of the relevant properties, which is often not present either.

In the following, we will explicitly look at the four categories of interest mentioned above (i.e., mapping structural concepts, mapping primitive types, mapping static and dynamic semantics). Each of them will form a separate subsection for the concrete bridges discussed in the next sections.

In general, conceptual mappings should strive for completeness. Both the structural concepts and the primitive types should be mapped to the best degree possible. If concepts are not mapped, information is potentially lost, or at least not available in the opposite space. With regard to correctness, we have already argued that the missing semantic formalization of many technical spaces renders proofs about mappings impossible. Nonetheless, if possible, conceptual mappings should preserve the semantics when bridging spaces. Additionally, conceptual mappings should be independent of particular applications to enable their reuse.

5.1.2 Language Mappings

A conceptual mapping between technical spaces, as discussed in the previous section, defines valid relations between the concepts of two technical spaces. Languages use (i.e., instantiate) the concepts of spaces. Consequently, if a language shall be used in another space, a corresponding language definition is required. We call the mapping between languages, conforming to the conceptual mapping, a *language mapping*.

The instance-of relation that holds between conceptual and language mappings is characterized by the degree of freedom that is put forward by conceptual mappings. Consider, for example, a conceptual mapping between the EMOF and the ontology space. In EMOF, metamodels are used to define data structures. Each metamodel contains a number of metaclasses. In the ontology space, ontology classes are used to define domain concepts. Thus, even though the conceptual mapping between both spaces may state that metaclasses must be mapped to ontology classes, it is not yet defined which metaclass needs to be mapped to which ontology class. This is the degree of freedom, which allows different instantiations (i.e., different language mappings) for the same conceptual mapping. One can also say that the conceptual mapping defines the set of all valid language mappings.

Having said that conceptual mappings leave options for concrete language mappings, one must add that building bridges does often include a mechanism to derive default language mappings. For example, given a metamodel in EMOF one can derive a default ontology, where all ontology classes have the same name as their corresponding metaclasses. This derivation of a default mapping is possible, because one does not need to map two existing languages, but rather one given language needs to be translated to a corresponding language specification in the opposite technical space. In many cases, the default mapping needs to be augmented with hand-crafted parts.

Similar to conceptual mappings, where we discussed the necessity to map structural concepts, primitive types, as well as static and dynamic semantics, language mappings raise the same questions, but on a lower metalevel. Instead of mapping the concepts that can be used to define languages, one does now map concepts defined by a concrete language. Again, the means to build up complex structures (e.g., defined references) must be mapped. If the mapping of primitive types was not completely covered in the conceptual mapping, this needs to be refined. This is often the case, because different languages require different mappings of primitive types. Last but not least, the semantics of the artifacts that conform to a language specification must be considered. Again, both static and dynamic semantics need to be mapped correctly. The latter is more important for language mappings, because concrete languages are often subject to execution and do therefore have dynamic semantics. This is different from the conceptual mapping, where the instances (i.e., language specifications) are rarely executed.

5.1.3 Technical Integration—Transformation, Adaptation

The goal of bridging technical spaces is to enable the use of tools from one space on data from the other. To achieve this, there are two different possibilities. First, one can transform data present in one space to the physical representation required by the tools in the other space. This approach will be called a *transformational bridge*. Second, one can adapt existing query language engines or the API used by tools in the target space to access the data of the source space directly. This method will be called an *adapter bridge*. The difference between the two kinds of bridges is also depicted in Fig. 5.3.

Adapter bridges can be used only if tools process data indirectly using an API or query language. If tools access data directly (e.g., by reading files from a disk), a transformational bridge must be used. Both styles of bridges have their pros and cons. To clarify which style is best to be used in which situation, we will investigate both kinds of bridges in more detail.

In [154], bridges are distinguished into *extractors* and *injectors*. The latter create models from software artifacts, while the former realize the opposite. Also, the term *projector* is used in [154] to denote both extractors and injectors. This differentiation is orthogonal to transformation and adaptation. It depends on the target space of a bridge rather than its technical realization.

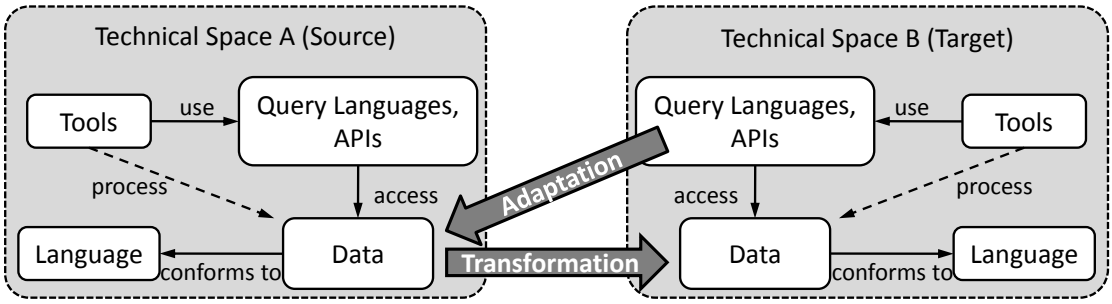


Figure 5.3: Transformation vs. adaptation to bridge two technical spaces.

Integration by Transformation

When integrating tools that employ different data formats, transformations can be used to convert from one format to the other. These transformations take the physical data format of one technical space as input and produce the format of another one as output. If bidirectional tool interoperability is aimed for, either multiple transformations or bidirectional ones are needed. As translations are executed monolithically, a batch-like processing of data is inevitable. Tools must be executed strictly in sequential order, where transformations may be placed between the execution of tools or tool runs.

One drawback of the transformational approach is the need to serialize (save) and deserialize (load) data whenever switching to another technical space is necessary. This can become a serious bottleneck if switching spaces is required often or if large amounts of data need to be transformed.

Furthermore, transformational approaches must deal with the concrete syntax of all involved artifacts. If there exist formal specifications of these syntactic representations, one can use dedicated tools (e.g., parser generators) to read and persist artifacts. If the knowledge about the concrete syntax is hidden (e.g., only the artifact-specific tools implement the functionality to access files), this can become a serious problem. In this case, the concrete syntax needs to be reverse engineered, which usually requires a large amount of work.

Integration by Adaptation

When using transformations to bridge technical spaces, each tool (or group of tools) uses its own physical representation of the information to be processed. To avoid such physical replication of data, an alternative solution is to implement an adapter that translates data instantly by forwarding data requests to a different data repository. Here, tools from the target space access the data of the source space using an API. Instead of replicating data, the data is directly presented to tools in the required form. In other words, the data repository interface is adapted to a repository of a different technical space.

Transformational Bridges	Adapter Bridges
⊕ feasible if tools access data directly	⊖ restricted to tools that use an API
⊖ replication of data	⊕ no replication of data
⊖ monolithic translation	⊕ incremental translation
⊖ sequential use of tools	⊕ concurrent use of tools

Table 5.1: Comparison of transformational vs. adapter bridges.

If tools use an API to access data, one can exchange the implementation of this API with a custom one which adapts the different representation dynamically. In this case, only knowledge about the concrete syntax within one technical space is needed. This can sometimes ease the creation of such an adapter-based bridge.

In addition, data that is accessed using an API can be shared in the sense that multiple APIs can be bound to the same data repository. This avoids the batch-like processing that is characteristic for transformational bridges. Tools from different technical spaces can access data simultaneously if respective adapters exist.

Comparison

When comparing the two styles of bridges that can be used to close the gap between two technical spaces, the drawbacks of one style can often be resolved by the opposite style. Consult Table 5.1 for the differences between the two styles.

While adapter bridges avoid replication, allow for on-the-fly translation and permit the use tools concurrently, they are restricted to applications where the involved tools use an API or query language to retrieve data. In contrast, transformational bridges are not restricted w.r.t. the tools in the target space. However, they force to transform monolithically and to replicate data.

The style to pick for a concrete scenario needs to be selected according to this criteria. If tools in the target space are sufficiently decoupled from the physical representation of data by the use of an API or query language, implementing an adapter bridge is a good choice. The integration of this kind of bridge is more tight than transforming physical data back and forth. If tools are strongly coupled to the physical representation of their data, a transformational bridge is the only option left. Despite of the drawbacks listed in Table 5.1, transformational bridges often perfectly serve their purpose. If there is no need to modify data concurrently and if the transformation of reasonable amounts of data is sufficiently fast, replication may be tolerable.

From a more global point of view, there are parallels between the two types of bridges between technical spaces and the two options to counter redundancy in software development (cf. Sect. 3.1). Adapter bridges avoid redundancy by tying tools to different data repositories. Transformational bridges create redundancy by representing data dif-

ferently, but need to synchronize the replicated pieces of data if changes are made. Thus, the same two principles can be applied both to bridge technical spaces and to deal with redundancy within a single technical space.

5.2 Existing Bridges for Technical Spaces

The problem of bridging technical spaces has been faced by many software engineers. Consequently, tools, procedures and standards exist to build bridges between common spaces. Examples for standards are Common Object Request Broker Architecture (CORBA) [166], Interface Definition Language (IDL) [167] and Web Services Description Language (WSDL) [168]. This section reviews a selection of existing approaches to bridge technical spaces and compares them with the generalization presented in Sect. 5.1.

We will restrict the presentation of existing bridges to the ones that are considered important in the context of our work. This particularly applies to the XML space, EMOF and Java, which are also frequently used in industrial applications. Similar bridges for Microsoft's modeling technology and EMF are also available [169, 170], but these are out of our scope.

5.2.1 XML—EMOF—Java

XML, EMOF and Java are three very popular technical spaces. XML was first published in 1998 [171] and has made its way into a countless number of applications and influenced many other standards. It can be considered one of the most widespread data exchange formats. EMOF is at the heart of the MOF standard [6] and therefore essential for many modeling activities. In particular, its implementation in the EMF, has made it the quasi-standard in object-oriented modeling. Java—a popular object-oriented programming language—is also widely used in industry.

To process XML data using Java programs, different options are available. The Document Object Model (DOM) [172] defines an API to access and manipulate arbitrary XML documents. Alternatively, the Simple API for XML (SAX) [173] can be employed to process XML data with the Java programming language. While DOM is a document-centric approach, SAX is event-oriented and allows to process XML documents as they are being read.

Besides the opportunities given by DOM and SAX, EMF provides functionality to bridge the XML space and the Java world using EMOF as connector. EMF includes an XSD importer that can convert schema definitions to EMOF models. This allows to process XML languages equivalent to EMOF-based modeling languages. EMF is also able to generate a set of Java interfaces that resemble the structures defined in an XSD document. These interfaces can be used to read and manipulate respective XML data. Thus, EMF provides a bridge between these three technical spaces. The

XSD Concept	EMF Concept	Java Concept
Schema	EPackage	Package
Simple type (union, list, anonymous)	EDataType	Class
Simple type with enumeration facets	EEnum	Class (or Enum)
Complex type	EClass	Class/Interface
Extension, restriction of complex types	eSuperTypes	Inheritance
Abstract type	Abstract EClass	Abstract Class/ Interface
Attribute	EAttribute/EReference	Field
Element	EAttribute/EReference	Field

Table 5.2: Partial mapping between XSD, EMF and Java concepts.

conceptual mapping between the language concepts employed by each technical space is summarized in Table 5.2.

Here, simple types are mapped to `EDataType`s and `EEnum`s. Each complex type corresponds to an `EClass`, extensions and restrictions yield inheritance relations. XSD attributes and elements are translated to `EAttribute`s and `EReference`s.

This conceptual mapping connects the language concepts of XML, EMOF and Java, as described in Sect. 5.1. Moreover, the concepts of these spaces are sufficiently similar to automatically derive corresponding language mappings. An XML schema can be translated to a metamodel and also to a set of equivalent Java classes. There is sufficient information to instantiate the conceptual mapping for a concrete schema.

This does not imply, that there are no variability points within the instantiation of the conceptual mapping. But, for all arising ambiguities (e.g., an element in the schema can be mapped to multiple alternative elements in a metamodel), a default choice is specified by EMF. EMF does derive default language mappings in the corresponding technical space automatically. Similar strategies can be observed for other bridges as well. For example, the bridge between relational databases and Java can derive a default set of Java classes to represent a concrete database schema (cf. Sect. 5.2.2).

The derivation of a concrete language specification for a technical target space from a given language follows the idea that bridging spaces technically should not be mixed with bridging them semantically. The data residing in one space is made available to another space by deriving a dedicated, equivalent target language. For example, if a schema for office documents shall be represented in EMOF, it is translated to a custom metamodel. It is not mapped to an existing metamodel for documents. We transfer data independently of a specific target language.

Relational Database Concept	Java Concept
Schema	Package
Table	Class
Table/Column	Inheritance
Column/Table	Field

Table 5.3: Mapping between concepts of relational databases and Java.

The bridges between XML, EMOF and Java that are provided by EMF are transformational bridges. One can access the same data using both XML-based, EMOF-based and Java-based tools by replicating XML data as Java objects. This is achieved by the use of different algorithms for serialization and deserialization. However, this approach does also imply, that tools cannot be used simultaneously, because serialization is required before switching the technical space.

5.2.2 Relational Databases—Java

Another, very important technical space are relational databases. The majority of enterprise applications rely on such databases to store and process business data. Since the interaction between relational databases and applications is based on standardized query languages (e.g., SQL), the actual representation of data in such databases is hidden from clients. Commercial database systems do not even provide means to access data directly—they use a proprietary binary format to persist information.

In the case of Java, multiple frameworks exist that help developers to access relational data within object-oriented Java programs. Low-level SQL APIs can be used to retrieve information from a database, but these do not convert between the concepts of the two spaces. Instead of returning objects (i.e., concepts of the Java language), row and column values (i.e., concepts of the relational world) are obtained. The resulting bridge is therefore quite weak in the sense that a large portion of the mapping is left to the user. Manual labor is inevitable in this case.

Other, more sophisticated frameworks for object-relational mappings can do better. For example, the Java Persistence API (JPA) [174] allows to specify declarative mappings between classes and tables, as well as fields and columns. These declarative mappings are then used to represent data from relational databases as objects and vice versa. Possible mappings between the concepts of the two technical spaces are shown in Table 5.3.

One can see that multiple valid mappings between concepts do exist here. For example, Java fields can be either mapped to columns or separate tables. This choice depends on the multiplicity of the field (i.e., collections are usually stored in separate tables), but can also be influenced by performance considerations. Similarly, classes that are connected by an inheritance relation, can either be represented by a single table or two

separate ones. In the first case, the columns that correspond to fields of the subclass are not used in rows that represent instances of the superclass. In the latter case, attributes of the subclass are stored in an additional table.

This conceptual mapping does leave space for variation. Concrete language mappings between database schemata and Java packages can still pick different options for the elements that need to be mapped. Additional options, which were not shown in Table 5.3, include the mapping between the primitive data types of both spaces. This variability is similar to the examples shown in the previous section. The instantiation of the conceptual mapping for a concrete language (i.e., a schema) can be performed differently. But, again default settings can provide a default instantiation result. For example, a default set of Java classes can be derived from a given database schema.

Bridges between relational databases and Java are usually realized as transformations. The bridges replicate data, because Java objects hold copies of the values stored in the database. The amount of redundancy that is introduced can differ from bridge to bridge. If a query language is used to retrieve and modify data, few portions of the overall data set (i.e., only the part that is of interest) is replicated. For object-relational mappings (e.g., the ones that can be built using JPA) the amount of replicated data may even be bigger since all values that are stored in rows and columns are copied to Java objects, which can be manipulated in memory and then be written back to the database. Even though the bridges often employ an API to access data, the functions of the API are used to read and store data. Java programs that operate on this data, do perform modifications on copies and not directly in the database.

5.3 Bridging Context-free Grammars and Object-oriented Modeling

Many artifacts that are used in software development have been traditionally specified using context-free grammars. Such grammars are both very well understood on a theoretical level and supported by a great amount of tools. Based on formal language theory important properties of context-free grammars, such as the complexity to parse their sentences, are known. Context-free grammars provide a reasonable trade-off between the complexity of the languages they can generate and the complexity required to handle the languages (e.g., to recognize correct sentences). The simplicity, the well understood formal grounding and the early availability of powerful tools to practically use them, made context-free grammars a widely used method to describe software artifacts. For example, many programming languages are based on context-free grammars.

With the advent of MDSD, metamodels, being another formalism to describe software artifacts, have gained acceptance and are starting to replace traditional syntax specification methods. Metamodels share some properties with context-free grammars, while they significantly differ in others. For example, metamodels are similar in the sense that

they provide a precise definition of all syntactically correct models. Thereby, a natural instance-of relation equivalent to the one between grammars and their sentences is established. To name a difference, metamodels describe graph structures, whereas grammars allow for the construction of strings and trees only.

Despite the differences between the two technical spaces spanned by context-free grammars and metamodels, bridging these spaces is an essential task within the scope of this thesis. It is important to unify the representation of the majority of artifacts used in productive development processes (i.e., code and models). Without bridging the two spaces, no synchronization method can be uniformly applied to all kinds of artifacts. Rather, multiple methods—one for each technical space—would be needed, which renders the creation of RTE systems more difficult.

Before discussing the concrete mapping between context-free grammars and object-oriented models, we will shortly recapitulate the formal definition of context-free grammars. We will use the definition from [175], which is as follows. A context-free grammar G is a 4-tuple $G = (V, \Sigma, R, S)$ where

- V is a set of non-terminals,
- Σ is a set of terminals, disjoint from V. The set of terminals is the alphabet of the language defined by the grammar.
- R is a relation from V to $(V \cup \Sigma)^*$ such that $\exists w \in (V \cup \Sigma)^* : (S, w) \in R$. These relations are called productions or rewrite rules. The asterisk represents the Kleene star operation.
- S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

From this formal definition, the central concepts of context-free grammars—terminals, non-terminals and rules—can be derived. For practicability reasons, we will use an extended version of the definition above, which allows advanced operators based on the Extended Backus Naur Form (EBNF) [176, 177].

The EBNF provides multiplicities (asterisk, plus and the question mark), which allow the definition of repeating and optional syntax without the need to introduce additional non-terminals. For example, the two productions $V_1 \to a$ and $V_1 \to \epsilon$ can be replaced by a single one $V_1 \to a?$. In addition, EBNF supports choices to inline non-terminals which represent alternative syntax. For example, the two productions $V_2 \to a$ and $V_2 \to b$ can be replaced by $V_2 \to a|b$. Furthermore, EBNF allows to group terminals, which essentially inlines a production. All these advanced operators can be translated to the simplistic productions defined above.

The concepts of object-oriented metamodels, in particular the ones that conform to EMOF, can be found in Sect. 2.1. In the following, we will present the required conceptual mapping in Sect. 5.3.1. This includes a discussion of mapping the structural

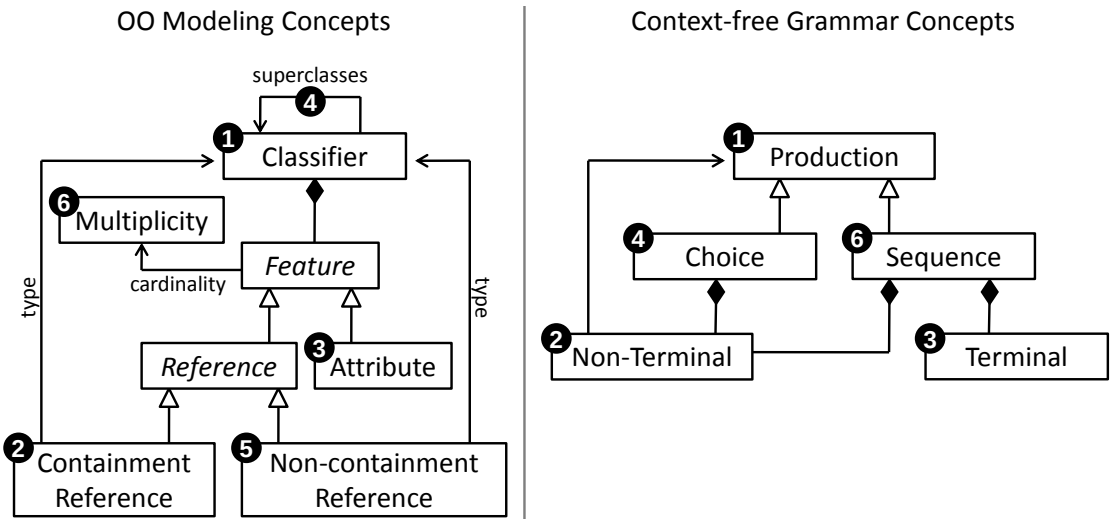


Figure 5.4: Conceptual mapping between models and context-free grammars.

concepts of both spaces, as well as an investigation on how to map primitive types. We will also look at the semantics that both spaces are accompanied with and the degree to which these are compatible. The conceptual mapping has been implemented in the tool EMFText [178]. We conclude our analysis of the established mapping in Sect. 5.3.2, where questions regarding the correctness and completeness of bridges between the two spaces will be answered.

5.3.1 Conceptual Mapping

Structural Mapping As outlined in the description of the general mapping problem of technical spaces (cf. Sect. 5.1.1), structural concepts are very important when transferring information from one space to another. Therefore, the structural concepts of both context-free grammars and object-oriented models are depicted in Fig. 5.4, as well as a potential mapping between them.

One can recognize concepts such as productions, terminals and non-terminals for the grammar space and terms like classifier, reference and attribute for the modeling space. Before mapping the concepts to each other, let us deepen the understanding of the concepts a little more.

First, models defined by modeling languages are composed of elements that in turn consist of attribute values and other contained elements. In addition, cross-references between elements can exist. Which elements are allowed is defined in a metamodel through classifiers. By defining containment references between classifiers, valid containment relations are defined. Which attribute values can be defined for an element is

declared by attributes in the metamodel. Classifiers are also connected through superclass relationships, which express exchangeability. Possible cross-references are defined by non-containment references.

Second, sentences defined by context-free grammars consist of a sequence of elements, where an element is either represented by a single symbol (i.e., a set of connected characters) or another nested sequence, which again consists of symbols and possibly other nested sequences. Valid parts are defined through productions. By referencing other productions through non-terminals in a sequence, the possible nesting of elements is defined. The symbols that may appear in a sequence are defined by terminals. Choices of different non-terminals can be defined to express exchangeability.

Based on [2], the following mapping can be derived for the core structural concepts of both technical spaces:

- Classifier—Production (1–1)
  Element types are defined through classifiers in one space and by productions in the other.

- Containment-Reference—Non-Terminal (2–2)
  The composition of elements from others is expressed by containment references on the metamodel side. Non-terminals that appear in a sequence, express a similar relationship on the grammar side.

- Attribute—Terminal (3–3)
  Attribute values are also part of a model element and can therefore be expressed by symbols.

- Superclass—Choice (4–4)
  The superclass relationship can be mapped to the alternative concept because both express exchangeability.

- Non-Containment-Reference—Terminal (5–3)
  For cross-references, there exists no direct correspondence in grammars. They can however be mapped to terminal symbols as well. In this case, the symbol represents an identifier that identifies the element to be referenced.

- Multiplicity—Sequence (6–6)
  Features of metaclasses can hold multiple values depending on their cardinality. In grammars, sequences can be used to express that elements occur multiple times.

A closer look at this mapping reveals that the structural concepts of models and context-free grammars are quite similar. Up to a single case—mapping non-containment references to terminals—all concepts have an exact counterpart in the opposite technical

space. Furthermore, all concrete concepts (i.e., all except `Reference` and `Feature`) can be mapped, which indicates that the mapping of concepts is complete.

The only discrepancy w.r.t. the structure of artifacts from both spaces, is the fact that both non-containment references and attributes are mapped to terminals (i.e., a non-injective mapping). This situation is triggered by the mathematical structures that form the bases of the two spaces. Models are based on graphs, whereas grammars use trees as primary means to construct complex structures. Trees do not allow arbitrary connections between nodes, whereas graphs do so. Non-containment references are exactly the connections, which are not allowed by trees. Therefore, no direct equivalent can be found in the context-free grammar space.

But, to represent such connections to other nodes in the grammar space, it is common practice to use terminals. For example, most programming languages refer to declared elements using identifiers. Even though there is no explicit support in context-free grammars to state that an identifier references certain other elements in the grammar, this connection is often part of the static semantics of the language defined by the grammar. In contrast, models use explicit references. For example, a reference to declared elements is stored in models, rather than using a symbolic identifier.

In summary one can say, that the formalism to build complex structures in one space can be resembled in the other.

Mapping Primitive Types After having established the mapping of the structural concepts, the relation of primitive types available in context-free grammars and models needs to be clarified. The former do have only one primitive type—strings. There is no explicit distinction between numbers, dates or booleans. The basic building blocks of all sentences are characters and sequences of characters. Models on the other hand are based on the MOF standard, which imports a set of predefined primitive types from the UML infrastructure specification [21]. This set includes `Integer`, `String`, `Boolean` and `UnlimitedNatural`[1].

Since grammars do use only one primitive type, whereas models allow multiple, a conversion is needed. The type of the concrete attribute that is mapped to a terminal determines the target for this conversion. For example, if a boolean attribute is mapped to a specific terminal, the values of this terminal must be transformed to either `true` or `false`—the possible values for boolean typed attributes.

Naturally, concrete conversions depend on the semantics of the context-free grammar at hand. As there is no information about the meaning of terminals in the grammar itself, the designer of a concrete bridge must decide how to convert string values of terminals to the primitive types used by attributes in models. One can say that the conversion of primitive types cannot be handled generically as the information required

[1] In contrast to `Integer`s, `UnlimitedNatural`s do not allow for negative values, but additionally include infinity, which is denoted by an asterisk.

to do so is not formally available. Converting primitive types is left to the designer of a concrete bridge between a grammar and a metamodel.

Mapping Semantics Besides the structural aspects that can be found in technical spaces, the semantics of artifacts that reside in such spaces is an important issue to consider when bridging spaces. Even when structures can be mapped from one space to another, this does not necessarily imply that two artifacts have equivalent meaning.

For the case at hand, the semantic mapping is rather narrow. Grammars, being primarily developed to describe syntax rather than semantics, do not have a complex semantics. The single important property that should be preserved by mappings of grammars to another space, are the rules by which productions can be expanded (i.e., the construction of sentences from a grammar). Informally these rules are as follows:

- Starting from the start symbol, any non-terminal on the right side of a rule can be replaced by the rule for this non-terminal. This rewrite procedure can go on until all non-terminals have been replaced by terminals.
- Choices in grammar rules must be replaced by a single option.
- Choices and non-terminals can be picked non-deterministically during replacement.

When looking at rewrite rules in the context of our mapping to the modeling space, the following observations can be made. First, metamodels do not have the notion of a start symbol. One can start the creation of a model by creating an instance of a random metaclass. In further steps, this instance can be extended by adding more instances and by adding references to them. Thus, grammars are more restrictive in the sense that they identify a particular top-level language element—the start symbol. Metamodels are less restrictive, because they allow to construct models starting from arbitrary, potentially multiple root elements.

The actual expansion of grammars and models is preserved by our mapping. Replacing a non-terminal by the right side of the respective production, is equivalent to the creation of an instance of the metaclass that corresponds to that production. Of course, the new instance must be referenced. In addition, choices which allow the definition of alternative syntax in grammar rules, were mapped to the inheritance relation between metaclasses. This mapping is also valid, since the semantics of subclassing is equal to the one of choices—one must pick exactly one out of the given possibilities (i.e., either an option or a subclass respectively).

Consequently, if grammar and metamodel concepts are mapped according to Fig. 5.4, the expansion of the grammar is parallel to the creation of a model. This is a nice property as it guarantees, that there is a valid textual representation for every model and that a model exists for every sentence of the grammar. The former requires, that the

model is constructed starting at the root metaclass (i.e., the metaclass that corresponds to the start symbol of the grammar).

If models have multiple root elements, a gap between the two spaces exists. However, this can be easily resolved. For example, one can introduce an artificial start symbol to the grammar, which allows to produce all the non-terminals that correspond to root metaclasses. Or, the other way around, an artificial start metaclass can be introduced that serves as superclass for all root metaclasses. In both cases, slight modifications of the grammar or metamodel are needed to establish a mapping. However, these modifications are very lightweight, which is why the start symbol problem is not severe.

5.3.2 Conclusion

Context-free grammars and models form a pair of technical spaces, which is highly relevant in the context of RTE. The former is the most widespread formalism to capture the structure of software artifacts formally. The latter is gaining significance with the advent of MDSD. The bridge that was conceptually developed above and implemented in the EMFText tool, allows to apply modeling tools—editors, analyzers, or transformation engines—to grammar-based artifacts. This is not only useful in the context of this thesis, where the main focus is RTE, but opens up many other application areas. For example, Reverse Engineering existing applications becomes possible [179].

Even though context-free grammars are only one existing technical space out of many, its flexibility and thereby its wide usage is striking. The vast amount of artifacts that can be described by context-free grammars makes this particular bridge especially important. It allows to represent different kinds of artifacts using a common syntax formalism (i.e., metamodels). Once all artifacts reside in the same technical space, synchronization procedures can be uniformly applied. Also, the technical and the logical translation (cf. Sect. 3.4.3) are strictly separated.

Besides the fact that the creation of this bridge allows many artifacts to be subject to RTE, one must be aware of the limitations of the mapping between the two spaces. When facing the concrete task of establishing a bridge between a grammar-based language and a respective metamodel, manual labor is often required. Depending on the complexity of the involved language, this may range from very little specification effort to enormous time consuming work that is needed to fill the gap between a grammar and a metamodel. The reason for requiring manual labor here, is the lack of important information in the grammar of the language at hand.

First, context-free grammars are based on typed trees. Thus, the first missing piece of information are the cross-links needed to establish a graph structure. This graph structure is often defined by the semantics of the language, but not necessarily captured formally. As a consequence, cross-links are part of the language, but no description of the links is readily available. Therefore, building a bridge involves a definition of the cross-references that form the graph structure of the artifact at hand.

Second, no primitive types are available in context-free grammars. From the perspective of a grammar, everything is a string. In contrast, metamodeling languages, such as Ecore, do provide a set of primitive types. Mapping back and forth between strings and these types is therefore inevitable. Again, this mapping must be defined manually.

Third, context-free grammars do not offer support for defining semantics. Usually, additional mechanisms (e.g., attribute grammars) are employed to formally define both the static and dynamic meaning of sentences. As the same lack of formal definition of semantics can often be observed for metamodels, the mapping between the two formalisms is left to the bridge designer. Moreover, there is often no way to check the correctness of such a mapping w.r.t. preserving semantics.

In general, one can summarize a few things that have been learned from building the bridge between grammars and metamodels. First, spaces that share a majority of similar concepts can be bridged even though manual labor may be required. The latter is mostly caused by the gap between the spaces—the information missing from one space or the other. Also, some properties of mappings are dictated by the relation of the concepts of the spaces. For example, the mapping of terminals to attributes (or non-containment references) is inevitable. The designer of a bridge can choose which attribute to map to, but the types of the two ends of the mapping are determined by the characteristics of the involved spaces. The size and nature of the gap between two spaces does also influence the completeness of a bridge between them. A rather small gap, as observed in this case, allows for more complete mappings, which in turn render concrete bridges more useful. In particular the fact, that the bridge between context-free grammars and metamodels is bidirectional, substantially increases its feasibility for many applications.

5.4 Bridging Models and Ontologies

To employ ontologies and associated tool machinery for the purpose of model synchronization, both a conceptual and a technical bridge is needed. This allows to treat models as ontologies and vice versa. The details of the conceptual differences that span this gap will be presented in Sect. 5.4.1. A conclusion about bridging ontologies and models can be found in Sect. 5.4.2.

5.4.1 Conceptual Mapping

Models and ontologies do have some commonalities, that sometimes lead to the wrong conception that both approaches are very close. Truth is that they share some properties, but do also expose huge conceptual differences. Even terms that are shared do have quite different semantics. Thus, a thorough analysis of the two spaces, their properties and semantics is needed and performed in this section. Discussions about representing models as ontologies can also be found in [180] and [181].

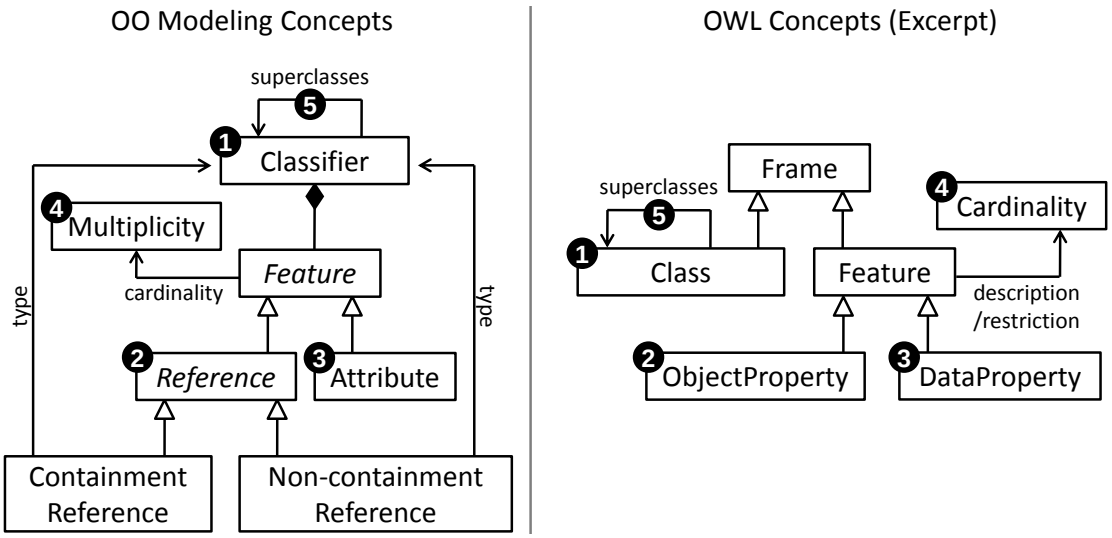


Figure 5.5: Conceptual mapping between models and ontologies.

Structural Mapping Similar to the comparison of context-free grammars and models, we will start with a figure that depicts important structural concepts of both technical spaces side by side. In Fig. 5.5, one can spot ontology concepts (e.g., DataProperty and ObjectProperty) on the right side. The concepts of models are the same as shown in Fig. 5.4 before.

The most obvious mapping can be found between the concepts classifier (in a model) and class (in an ontology) (1). Both denote sets of entities that share common properties. However, this mapping must be considered very carefully from the perspective of semantics, which we will do later on in this section. For the time being, let us assume these two concepts to be equivalent to a reasonable extent.

The next mapping can be established between Reference and ObjectProperty (2). Both can be used to model relations between things. Again, even though both are not exactly equivalent w.r.t. their semantics, we find them close enough to map them to each other. A similar relation holds for Attribute and DataProperty (3). Here, both concepts denote properties of things, which have a primitive type rather than an object (i.e., an instance of a complex type) as value. Besides the fact that the two pairs of concepts do not have the exact same semantics, it is also important to point out that references and attributes are owned by an identifier, whereas DataProperties and ObjectProperties are not. Thus, the latter two can exist independent of a class in an ontology. OWL does not have an explicit distinction between contained individuals and non-contained ones. This is why both types of references (i.e., containment and non-containment references) are mapped to ObjectProperties.

Ecore Concept	OWL Concept	Possible Caveat
EFactory, EOperation, EParameter	no mapping	ignored
EPackage	OWLOntology	inverse hierarchy
EClass	OWLClass	non-exclusive instanceof
EAttribute	OWLDatatypeProperty	name clash / qualification
EReference	OWLObjectProperty	name clash / qualification
EDatatype	RDFSDatatype	straightforward
EEnum	OWLDataRange	straightforward
EEnumLiteral	RDFSLiteral	straightforward
EAnnotation	RDFSLiteral	straightforward

Table 5.4: Mapping Ecore concepts to OWL (from [181]).

Similar to EMOF, OWL allows to restrict the cardinality of properties. Such restrictions can be defined by adding a data restriction component of type cardinality to the property axiom. Cardinality restrictions can specify minimal, maximal and exact cardinalities, exactly as in EMOF. Hence, the two concepts can be mapped to each other (4).

Besides the purely structural concepts of Ecore, there are also some concepts which cannot be mapped to the ontological space and some other concepts which are not used to compose structures, but ease modeling (e.g., enumerations and annotations). A general template for mapping these (additional) concepts and the ones shown in Fig. 5.4 has been proposed in [181] and is summarized in Table 5.4.

As indicated in the third column of Table 5.4 there are problems caused by the conceptual differences. For example, Ecore packages can access their parent package, while ontologies can only access imported ontologies. Also, there is a mismatch between the exclusivity of the instance-of relation, which will be discussed shortly. For attributes (or data type properties, respectively) and references (or object properties, respectively) name clashes can occur, because properties are globally visible in ontologies, while attributes and references are local to their declaring class. Mapping data types, enumerations, enumeration literals and annotations is straightforward.

The structural mappings shown in Fig. 5.5 and Table 5.4 do cover all essential concepts of EMOF, but not nearly all of OWL. This is caused by the completely different goals of the two languages. EMOF was designed to provide a minimal set of terms that can be used to specify object-oriented models. On the contrary, OWL was built to allow the modeling of arbitrary knowledge for potentially many domains. Thus, OWL provides more concepts than EMOF.

Mapping all concepts of OWL to EMOF is out of the scope of this thesis. This is not only because a semantically correct mapping is probably not possible, but because

few artifacts related to software development reside in the ontology space. Rather, tools that are available in this space are of interest to us. Thus, the most important goal of building the OWL-EMOF bridge is to represent models as ontologies and not the other way around. An analysis of the completeness of this bridge (cf. Sect. 5.4.2) will therefore focus on the coverage of the EMOF part.

Mapping Primitive Types Both OWL and EMOF have a set of built-in primitive data types. OWL relies on a subset of the data types defined by XML Schema [182], whereas EMOF uses the primitive data types defined in the UML infrastructure [21]. In addition, Ecore—the implementation of EMOF used in the context of this thesis—provides more data types than the ones defined by EMOF. A comparison of all data types can be found in Table 5.5.

In a strict sense, only the data types xsd:string, xsd:integer and xsd:boolean can be mapped from OWL to EMOF. A second set of data types can be mapped to Ecore, because it is available there, but not in the EMOF standard. Among this group are data types such as xsd:float, xsd:double, and xsd:dateTime. The last column of Table 5.5 indicates which mappings are not complete. A mapping is marked as incomplete if one of the two data types that are mapped to each other, has a bigger range than the other one or a slightly different meaning. Such incomplete mappings must be treated with special caution as they are not bijective. A last class of data types (e.g., xsd:gMonthDay) cannot be mapped at all, because there is no equivalent in EMOF.

Because of the set of built-in types for OWL and Ecore, this mapping is more complex compared to the simplistic handling of primitive types while bridging context-free grammars and models (cf. Sect. 5.3.1). But, here the mapping of data types can be performed already on the conceptual level, independent of concrete ontologies or metamodels. This reduces the effort, when building a bridge of this type. For the context-free grammars an individual mapping of string—the only available built-in primitive type there—needed to be defined for each concrete bridge individually.

However, one must be aware of the completeness criterion indicated in the last column of Table 5.5. If artifacts are modified both in the ontological space and the modeling space, situations may occur, where transferring artifacts to the opposite space is not possible anymore. This does especially apply when data types have different ranges. For example, if the value of a DataProperty of type xsd:unsignedInteger is changed such that the new value cannot be represented using a signed integer, the individual cannot be transferred to the modeling space anymore. Thus, one must either avoid such edit operations or define custom mappings, which resolve this problem (e.g., by mapping xsd:unsignedInteger to String).

Mapping Semantics Both models and ontologies do have the notion of concepts and instances. The former is called *class* in both worlds, while the latter is referred to as an

EMOF/Ecore data type	OWL data type (defined by XML Schema)	Complete Mapping?
String	xsd:string	yes
String	xsd:anyURI	no
String	xsd:normalizedString	no
String	xsd:token	no
String	xsd:language	no
String	xsd:NMTOKEN	no
String	xsd:Name	no
String	xsd:NCName	no
Boolean	xsd:boolean	yes
Integer	xsd:decimal	no
Float (Ecore only)	xsd:float	yes
Double (Ecore only)	xsd:double	yes
Integer	xsd:integer	yes
Integer	xsd:positiveInteger	no
Integer	xsd:nonPositiveInteger	no
Integer	xsd:negativeInteger	no
Integer	xsd:nonNegativeInteger	no
Long (Ecore only)	xsd:long	yes
Long (Ecore only)	xsd:unsignedLong	no
Int (Ecore only)	xsd:int	yes
Int (Ecore only)	xsd:unsignedInt	no
Short (Ecore only)	xsd:short	yes
Short (Ecore only)	xsd:unsignedShort	no
Byte (Ecore only)	xsd:byte	yes
Byte (Ecore only)	xsd:unsignedByte	no
Date (Ecore only)	xsd:dateTime	yes
Date (Ecore only)	xsd:time	no
Date (Ecore only)	xsd:date	no
Date (Ecore only)	xsd:gYear	no
n/a	xsd:gYearMonth	n/a
n/a	xsd:gDay	n/a
n/a	xsd:gMonth	n/a
n/a	xsd:gMonthDay	n/a
ByteArray (Ecore only)	xsd:hexBinary	yes
ByteArray (Ecore only)	xsd:base64Binary	yes

Table 5.5: Comparison of primitive data types for EMOF/Ecore and OWL.

object in modeling and an *individual* in ontologies. The relation between instances and classes is called *instance-of*. Classes denote sets of instances (i.e., objects or individuals). In the ontological world, classes are interpreted as elements of a taxonomy. Thus, there exists a generalization (or a specialization, respectively) relation between classes. While this is present in modeling too, ontologies put a less strict semantics on the instance-of relation. Individuals can have an instance-of relation with multiple classes even if no generalization relation is present. In the modeling world, each object is said to be instance of one class only. To be more precise each object can only be instance of two classes if both are related by a generalization relation. Classes are implicitly considered to be disjoint, while this is not necessarily the case for ontologies.

A related difference w.r.t. the semantics of OWL ontologies and object-oriented models is caused by the *Unique Name Assumption*. While different names in models denote things with different identities, this assumption does not hold for OWL. Here, it is perfectly valid to give multiple, different names to the same entity.

These first differences do already reveal the most important conceptual mismatch between modeling and the use of ontologies. The former do have implicit semantics (e.g., the disjoint classes constraint), while the latter do explicitly declare the meaning of the concepts formally. For example, the knowledge that classes in modeling are considered to be disjoint if they are not related by generalization, is captured in modeling tools only. To transfer models to the ontology world, all implicit semantics must be made explicit. Otherwise the meaning of the involved models will not be preserved.

In addition to the semantic differences between models and ontologies, there are also ontological concepts without equivalent counterparts in the modeling world. For example, set-theoretic operations like `unionOf`, `complementOf` or `intersectionOf` are not supported in models. Likewise there are modeling concepts that cannot be represented in ontologies (e.g., operations and parameters). In the context of RTE, these modeling concepts are less important, since one cares mainly about the data represented by the involved models (i.e., objects, attributes and associations).

Probably the most drastic difference can be found in the semantics of missing data. Most modeling languages have a prescriptive nature (cf. Sect. 2.1.1) and therefore consider missing information to be wrong. For example, objects can only populate properties that are defined in their corresponding class. Properties that were not defined are assumed to not exist. Ontologies on the other hand follow a descriptive approach and consider missing data to be unknown (i.e., potentially correct). An individual can therefore indeed have properties that are not defined by its class. This difference is often denoted as *open world* and *closed world* assumption.

5.4.2 Conclusion

The bridge between OWL and EMOF discussed in this section allows to apply tools and methods from the ontological space to models. In particular the use of reasoning

technology on models and metamodels is now possible. Or, as Kappel et al. put it in [181], models have been *lifted* to the level of ontologies. The term lifting does already imply, that the bridge at hand is not symmetric. It allows to treat models as ontologies, but not necessarily the other way around. Models that are based on the EMOF standard, use a very restricted set of concepts—the essential modeling concepts. They are also less expressive than ontologies. The proposed mapping allows to transfer important properties of models to the ontological space, but it is rather incomplete when it comes to representing ontologies as models.

The main motivation for building a bridge toward the ontology space is to employ reasoning tools for model synchronization. One can say that w.r.t. this goal, the established bridge is quite useful. However, it is not as complete as the bridge between context-free grammars and models. The latter can be easily reused for other tasks, which is not necessarily the case for the OWL-EMOF bridge. But, there are serious reasons for the observed incompleteness. OWL and EMOF span a much wider gap than grammars and models. This is due to the semantic differences between the two spaces.

The main problems that render building the OWL-EMOF bridge so complex are twofold. First, the amount of concepts defined by OWL outnumbers the ones provided by EMOF. Thus, things provided by OWL cannot be reasonably mapped to EMOF. Second, even though both OWL and EMOF are designed to model domain knowledge, each of them follows a completely different paradigm with regard to knowledge representation. EMOF is based on prescriptive modeling, where one exactly defines what is valid and what is not. This assumes complete knowledge about the modeled subject. In contrast, OWL is focused on descriptive modeling. Here, one states what is known and leaves other facts unspecified. Unspecified (i.e., missing) facts are not assumed to be false. Rather, anything that is not specified may actually be true or exist. The fact that one does not make statements about things is basically interpreted as unknown fact.

This difference in the modeling paradigm that forms the basis for the two technical spaces, causes most of the problems while bridging OWL and EMOF. In Chap. 7, where this bridge is employed in the actual synchronization of artifacts, we will further discuss how the gap between the two paradigms can be resolved.

Looking back at the complex and incomplete mapping between OWL and EMOF, one can observe some typical problems encountered when bridging spaces that are not particularly close. The bridge between context-free grammars and models required some additional work to obtain missing information (e.g., to determine cross-references to form graphs from trees). But, this integration was possible and also complete to a much higher degree. For OWL the case is different. The mismatch between the number of concepts, as well as the different underlying paradigms, causes an incomplete, yet useful bridge. However, this highlights an important issue, which is the criticality of selecting one technical space over another. Once a decision has been made to use artifacts or tools from a particular space, there is often no turning back. This is especially problematic, if the chosen space does offer concepts that cannot be found anywhere else.

5.5 Bridging Role Modeling and Object-oriented Modeling

Roles have been introduced by Halpin as means to encapsulate the behavior of objects within a specific context [183]. The tool integration approach that will be presented in Chap. 8 defines data repositories and data representations as such a context. There, role models are used to enable tools to work on arbitrary data. Since we aim at using models based on the EMOF standard as primary type of artifact, the question is how object-oriented modeling and role modeling can be integrated. This entails both a conceptual mapping and a technical realization that will serve as a basis to evaluate our tool integration approach. Before we dive into the specifics of object-oriented modeling and role modeling, we will briefly recapitulate their most important properties.

Object-oriented metamodels consist of classes, attributes and references. Classes can inherit from each other. Attributes have a primitive type, while references are assigned to complex types (i.e., other classes). Both attributes and references can have multiplicities that specify the amount of values (or objects, respectively) they can hold. Classes may also own operations, which specify behavioral aspects. However, for synchronizing models only data is considered. Thus, operations can be omitted.

Common standards for object-oriented metamodels do define more properties for classes, attributes and references. For example, classes can be abstract, attributes can be derived or references can be tagged as containment. For the time being, these additional properties will only be analyzed if they have implications for our work.

Role types and role models can be considered as an extension to object-oriented modeling [37]. Role types specify a relation between one or more classes. For example, the role type `Father` relates a man and his child. Roles, being instances of role types, can be attached to objects (e.g., a man becomes a father when birth is given to his child). In this case the man object is said to *play* the father role. Conceptually, classes and roles can be distinguished according to their rigidity [39]. Classes are rigid because they can exist on their own, while roles cannot, because they depend on an object to be attached to. For example, while a man can exist without being a father, the father role needs to be bound to a man. When the man deceases, its father role is inevitably lost. One can also say that classes are sets of things that carry an identity, while roles can only be identified by the identity of their players.

In this section we will analyze the conceptual relation between object-oriented models and role models (Sect. 5.5.1). Also, we will review the completeness of the established mapping between role models and object-oriented models in Sect. 5.5.2.

5.5.1 Conceptual Mapping

Similar to the mappings presented before and as proposed in the general procedure to establish bridges between technical spaces, we will now look at mapping structures, primitive data types and semantics. Emphasis is mainly put on the first and third point.

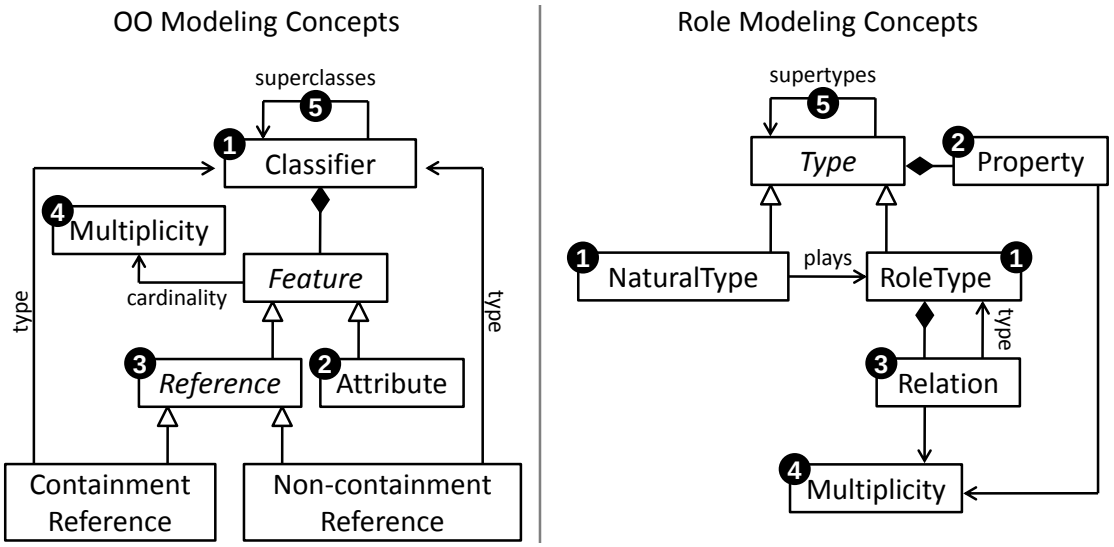


Figure 5.6: Conceptual mapping between object-oriented models and role models.

This is caused by the non-existence of a standardized metamodeling language for role models. Thus, we base our mapping on the assumption that role models and object-oriented models do share the same set of primitive types. As role models are an extension to object-oriented models, this assumption is reasonable.

Structural Mapping To map purely object-oriented models to role models, the mapping depicted in Fig. 5.6 is proposed. One can find both the object-oriented concepts (e.g., Classifier, Reference or Attribute), as well as the concepts of role modeling (e.g., NaturalType or RoleType).

The most simple mapping can be established between natural types and classes. In object-oriented modeling, classes are used to represent both rigid types (i.e., natural types) and non-rigid types (i.e., role types). No distinction is made between the two. Thus, representing natural types as classes is a reasonable option (Mapping (1) in Fig. 5.6). Both share the same meaning w.r.t. their usage in models.

Role types capture relations between classes. Thus, one could map them to references in object-oriented models. However, references in EMOF can connect exactly two classes only. If a role type establishes a relation across more than two types, it cannot be mapped to a single reference anymore. For example, consider the role type `TeamLeader` in Fig. 5.7, which relates one person—the leader—with multiple other persons—the team members—and a company—the one the team works at. Such a collaboration cannot be modeled with a single reference in EMOF. Thus, we map role types to classifiers, as these allow to model n-ary relations.

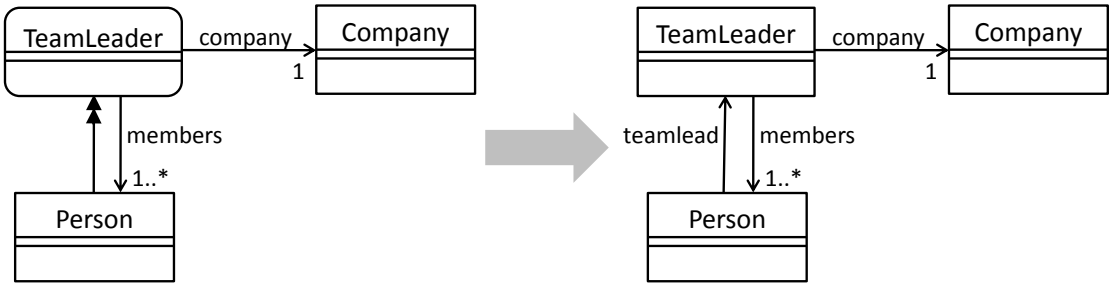


Figure 5.7: Example mapping of a role type to a class.

An example for mapping a role type (i.e., `TeamLeader`) to a class is shown in Fig. 5.7. Basically, the role type is replaced by a class and the `playsA` relation is replaced by a reference. Note that the multiplicities of the relations in the role model must be resembled by the ones in the object-oriented model. As the role model allows teams to have arbitrary many members, the object-oriented model must permit the same.

Types in role models—both natural types as well as role types—can have properties, similar to classes. Properties are equivalent to attributes in the sense that they must have a primitive type. Thus, mapping attributes to properties seems to be a natural choice (Mapping (2) in Fig. 5.6). References and relations do both connect types (i.e., classes, role types and natural types). They share the same semantics and are therefore natural counterparts (Mapping (3) in Fig. 5.6).

Handling inheritance between classes is straightforward, since both role modeling and object-oriented modeling, the former being an extension of the latter, provide this concept with exactly the same semantics. Thus, all inheritance links between types (i.e., either role types or natural types) can be represented by links between the respective classes (Mapping (5) in Fig. 5.6).

In total, the following mappings can be established between the concepts available in role modeling and object-oriented modeling:

- natural types can be mapped to classes (1),
- role types can be mapped to classes (1),
- properties can be mapped to attributes (2),
- relations can be mapped to references (3),
- inheritance is represented identically (5).

Mapping Primitive Types In the scope of this thesis, we will consider role models as extension to object-oriented models in the sense that we will add the concept of role

types to the concepts defined by the EMOF standard. Thus, both our role models and object-oriented models share the same set of primitive types. A mapping is therefore basically the identity function.

Mapping Semantics Even though roles seem to be a very clear concept at first glance, there is quite diverse interpretations about their semantics. A detailed comparison of the different aspects of the semantics of roles can be found in [184]. This does also include a list of references to the many published interpretations w.r.t. all the aspects. In the following, we will stick with this semantics (criteria according to [184]):

1. roles are types (i.e., they have their own properties and behavior),
2. roles do not depend on relationships (i.e., a role can be attached to a single natural type also),
3. objects can play multiple roles simultaneously,
4. roles can be dynamically acquired and abandoned,
5. role acquisition can be restricted (i.e., one role may only be acquired after playing another one),
6. unrelated types can play the same role,
7. roles can play roles,
8. roles can be transferred (from one object to another),
9. object states can be role specific,
10. object features can be role specific,
11. roles restrict access,
12. roles may share features or behavior,
13. objects share their identity with their roles.

For the purpose of binding modeling tools to arbitrary data repositories and data representations (cf. Chap. 8), where we are interested in the information represented by objects and roles (i.e., the static aspect of a modeled structure), the dynamic semantics of roles is not of utter importance. Thus, the criteria 4 and 8 can be neglected.

To check whether all other semantic criteria are respected by the presented mapping, one must keep in mind that object-oriented modeling subsumes both natural and role

types as classes. Thus, the question is whether the semantics of the concept class is compatible with the semantic properties listed above.

Criterion 1 states that role (and natural) types are full-fledged types, which is also true for classes. Criterion 2 is also fulfilled by classes, since these can be attached to a single other class using a reference anyway. Furthermore, classes can have references with an unlimited upper bound, which allows to reference multiple objects. This is equivalent to criterion 3. The restriction of role acquisition (criterion 5) is similar to restricting the addition of references in classes. While this is usually implemented in the code that handles reference additions and not modeled, the concept is present in object-oriented models as well. Criterion 6 is equivalent to having multiple classes that reference the same class. Roles playing roles (criterion 7) is simply captured by the fact that references between classes are not restricted w.r.t. the types they connect.

The next criteria (9 and 10) are more subtle, since objects on their own cannot behave different in specific contexts. However, if objects are encapsulated (e.g., by the use of delegation), different states can be observed and different features can be implemented. The specific observed behavior or obtained data does then depend on the concrete delegating object. Thus, this kind of semantics can be established both in object-oriented models as well as in role models. The same applies to criterion 11 with the same argument (restriction is possible using delegating objects).

Criterion 12 (sharing features and behavior among roles) is about inheritance between roles, which is along the lines of class inheritance and therefore compatible as well. The last criterion (sharing identity between roles and naturals) is more subtle. Since we map both role types and natural types to classes, the question is whether two objects belonging to two different classes can share the same identity. In EMOF, this is not possible, because classes denote disjoint sets of objects unless the classes are connected by an inheritance relation. However, to establish shared identities, one can use a dedicated procedure to check objects for equality. If this procedure (e.g., an operation `isSame`) implements shared identities, criterion 13 can be fulfilled.

One can see that the extension of object-oriented models to support roles is quite natural from a semantic perspective, if roles are defined as stated by the 13 criteria above. If the semantics of roles differs, which is often the case according to the context in which role modeling is used, a mapping would probably look different. For the scope of this thesis we will stick with the mapping and the semantics informally defined above, since both are reasonably compatible.

5.5.2 Conclusion

In this section, we discussed the mapping of role models to object-oriented models. First, the mapping of the structures found in object-oriented metamodel to role models needed to be clarified. The result of this investigation is that such a mapping is possible, but leaves certain degrees of freedom. For some mappings one can choose between different

possibilities. To correctly pick one of these choices, both the semantics of the language that is subject to the mapping and the RTE scenario at hand need to be considered. The former is particularly important to decide which types need to be modeled as rigid types (i.e., as natural types) or as non-rigid types (i.e., as role types).

We have also analyzed the proposed mapping w.r.t. its semantics. As it turns out, given the interpretation of roles used in this thesis, the mapping yields semantically compatible domain models. If a different interpretation of role concepts is used (e.g., by choosing another implementation for roles), this analysis needs to be revised.

5.6 Summary

Having identified the fact that software artifacts reside in different technical spaces as one of the blocking factors to build RTE systems, this chapter investigated on how to build bridges between such spaces. We have seen that a technical space consists of a set of concepts to define languages and tools to operate on them [154]. At a closer look, it turned out that every technical space entails artifacts to capture data, languages to formally describe artifacts and language concepts to likewise describe languages.

To bridge two technical spaces, it is required to map the concepts used to define languages. We have seen multiple examples for such mappings that varied in completeness depending on how wide the gap between the spaces was. We have also seen that mapping language concepts is not sufficient. Often, these mappings are templates that do still offer some variability w.r.t. mapping concrete languages. Thus, mappings on the language level are instances of the conceptual mapping between technical spaces.

Since no technique is known to derive conceptual mappings automatically, they must be created manually. Experts who know the semantics of the concepts of each technical space, must create a mapping to conceptually capture how data can be translated from one space to another. To actually perform such translations, a technical integration is needed that implements the conceptual mapping. Such integration can be performed by the use of transformations or adapters. The former replicate data to create a representation that can be used in the opposite space. In contrast, the latter adapt query languages or APIs to bridge spaces.

In this chapter we have investigated on multiple bridges and analyzed their conceptual mappings. Building bridges between context-free grammars and metamodels (cf. Sect. 5.3) turned out to require manual work to fill gaps between the two spaces. Nonetheless, many software artifacts are specified by grammars. Thus, these bridges enable many opportunities for RTE systems. Bridging the ontology and the modeling space (cf. Sect. 5.4) was more complex, mostly because these two spaces base on fundamentally different paradigms w.r.t. the interpretation of their artifacts. Finally, the bridge between role models and pure object-oriented models (cf. Sect. 5.5) was the easiest to build. This can be explained by the fact that we consider role modeling as an

extension of object-oriented modeling. Consequently, the gap between the two spaces was limited to the additional concepts of role modeling.

All the examples of bridges between different technical spaces have shown that it is possible to transfer data from one space to another, at least to a reasonable extent. We have also seen, that the translation required to do so, can be quite complicated. This strengthens our hypothesis that transferring data from one space to another must be cleanly separated from synchronizing data within one space. Both tasks are complex in their own respect. Mixing the two up, will make things even more difficult.

The bridges presented in this chapter aim to substantially ease building RTE systems. They allow to reuse RTE approaches present in one technical space for artifacts residing in different spaces as well. Also, the bridges can be reused independently from concrete applications or tools. This is substantially different from building individual adapters or transformations to access data that is represented differently, which is often common practice. In summary, the bridges presented in this chapter and the thorough analysis of their underlying mappings widen both the applicability of the subsequent approaches to RTE, but do also enable other tasks in software engineering to benefit from systematic translations between different technical worlds.

6
Backpropagation-based Round-Trip Engineering

In this chapter, we present a generic approach to RTE that is based on two basic observations. First, there are model transformations, which are not injective. In other words, one cannot determine a unique source model for a target model w.r.t. the transformation. For instance, multiple valid source models may exist. Second, it is often difficult to compute inverse transformations although they exist. That is, even if a transformation is bijective, deriving the opposite transformation for a unidirectional transformation can be problematic (cf. Sect. 6.1 for a detailed analysis).

The basic idea of our approach is therefore to restore global consistency across models (cf. Sect. 2.5) without computing inverse transformations. Instead, we propose to reuse existing unidirectional transformations and suggest to resolve inconsistencies by translating individual changes that are applied to target models to changes in the source models. We do not transform models, but model changes.

Translating individual changes instead of translating whole artifacts proved to be useful for specific RTE problems [107, 139, 147, 185]. Also, the work of Antkiewicz and Czarnecki on the design space of synchronization systems [186] mentions synchronizers that are based on change translation. This inspired us to generalize this idea and apply it to other application areas. For example, in [187] we transferred this approach to ISC and implemented it within Reuseware [188]—a practical model composition tool. In parallel to our investigations, a formal definition of change propagating RTE systems was presented by Hettel et al. in [156], but an application of these definitions to practical RTE problems was missing.

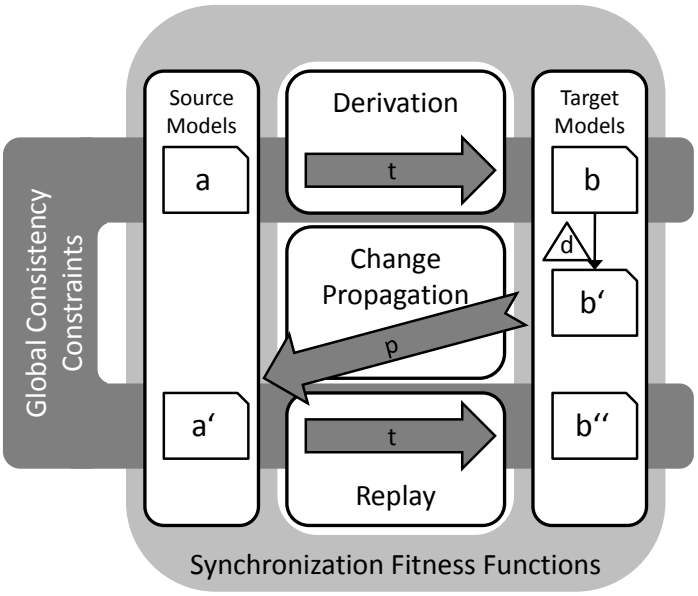


Figure 6.1: Backpropagation-based round-trip approach.

Thus, there was some evidence for the feasibility of change propagation both from a practical point of view as well as on a theoretical level. However, it was neither clear whether the similarities of the practical approaches could be captured by a generic framework, nor had the theoretical works been applied to complex practical problems. Thus, the limitations of the idea itself and the formal description were not known. This further encouraged us to pursue this idea and investigate how change propagation could help to solve practical RTE problems in a more generic way.

A sketch of the basic idea of change propagation is depicted in Fig. 6.1. Here, we can find a source model a and a target model b, which is *derived* using a unidirectional model transformation t. If a change d is applied to model b, we obtain a modified target model b', which may violate global consistency constraints. If it does, we need to restore global consistency by adjusting the source model accordingly. This yields a modified source model a'. Then, the transformation t is executed again to obtain a new globally consistent target model b''. We call the repeated execution of the transformation *replay* and the new target model an *echo model*. The latter emphasizes the fact that this model reflects the impact of adjusting the source model.

The main questions that need to be answered to implement this idea are the following. First, when and how can we derive source changes that correspond to target changes? Second, how can we ensure that the target model obtained after the replay step reflects the intentions of developers? Third, which RTE scenarios can be handled using the approach at hand? Answering these questions is subject of this chapter.

Main contributions of this chapter are

C6-1 a novel approach to RTE based on backpropagation and replay that eliminates the need for the computation of inverse transformations,

C6-2 a clarification of the role of traceability for RTE,

C6-3 an analysis of the applicability of the approach depending on the employed model transformation formalism,

C6-4 an analysis of this approach w.r.t. its advantages and disadvantages over related methods such as bidirectional model transformations.

We present our approach according to the following structure. First, the motivation for the approach will be recapitulated in Sect. 6.1. Second, a brief overview and some general definitions are given in Sect. 6.2. Third, an example is introduced in Sect. 6.3, which will be used subsequently. Fourth, details of the backpropagation-based RTE approach are presented in Sect. 6.4. This entails how to apply the approach to a new RTE scenario. We discuss the approach in Sect. 6.5 and focus on its limitations w.r.t. practical applicability. Section 6.6 concludes this chapter with a short summary.

6.1 Motivation

The problem analysis conducted in Chap. 3 revealed that finding inverse transformations can ease RTE, but is often difficult to realize. For example, consider a transformation that composes large models from small model fragments. If one can find a transformation that is able to decompose models (i.e., a inverse transformation), synchronization between model fragments and the composed model is accomplished. However, there may be multiple valid decompositions, because the same composed model can be built from different pieces. Thus, no single correct inverse does exist. In [189] it is shown that even simple transformation tasks cannot be realized using injective transformations.

Moreover, even if we assume a scenario where there is exactly one valid decomposition, there are further obstacles. The existence of a single decomposing transformation does not imply that this transformation can be computed. If one cannot give an automatic procedure to find inverse transformations even though they exist, inverses must be derived manually. In the context of model composition, this means that one must manually provide a decomposition specification for each composition. Thus, the first two main observations that motivate our backpropagation-based approach to RTE are:

O6-1 There are RTE scenarios where inverse transformations do not exist.

O6-2 There are RTE scenarios where inverse transformations do exist, but cannot be derived automatically.

In addition to these general observations, we recognized that existing solutions to specific RTE problems share similar strategies. For example, Chalabine and Kessler [147] presented an approach to provide RTE functionality in the context of Aspect-Oriented Programming (AOP). They developed a framework to propagate changes across woven programs, program cores and aspects by tracing tree compositions. Instead of using a decomposing transformation, they propagate changes individually.

Becker et al. [185] employed change propagation to integrate a set of proprietary tools. They did also employ links between related artifacts and rules to define how changes are propagated. However, the limits of their approach were not clear. The properties of the relations between the artifacts that must holds to enable the approach were not studied.

Even earlier, the database community followed the same strategy to synchronize relational views with underlying tables [139]. Again, changes were translated, but no inverse transformations were required. The work on updatable database views was continued by Bohannon and Pierce [190], who introduced the notion of *lenses* as a means to propagate changes. Even though, the lenses in [190] did not translate changes, but whole database states, the authors suggested investigations in this regard as future work. Thus, we can retain another important observation:

O6-3 Solutions to practical RTE problems employ change propagation.

Driven by the idea that propagating individual changes could be useful for model synchronization too, we successfully transferred the basic procedure to model composition [187]. This did create further demand for a general framework which covers all use cases that were studied in isolation. The question which properties enable the application of RTE support based on change propagation became fundamental. We had seen instances of the general idea in different scenarios, but a thorough understanding of the interrelations was missing. Thus, two research questions emerged:

Q6-1 (Full Generality) How can specialized RTE solutions be derived from a generic solution?

Q6-2 (Applicability) When is backpropagation-based RTE applicable?

Since our goal is to establish RTE support for MDSD environments, another observation pushed the motivation further—the sequential enrichment of models. In order to complete abstract system specifications in MDSD processes, domain knowledge and technical details are added and a chain of models is established (cf. Fig. 6.2).

Enriching models can be performed either by using automatic transformations or by applying manual changes. While both are different in terms of their execution, they can be conceptually treated the same. An automatic transformation entails a predefined set of change operations, while manual editing can be conceived as creating a set of changes. Still, from an abstract point of view, both do apply changes to models.

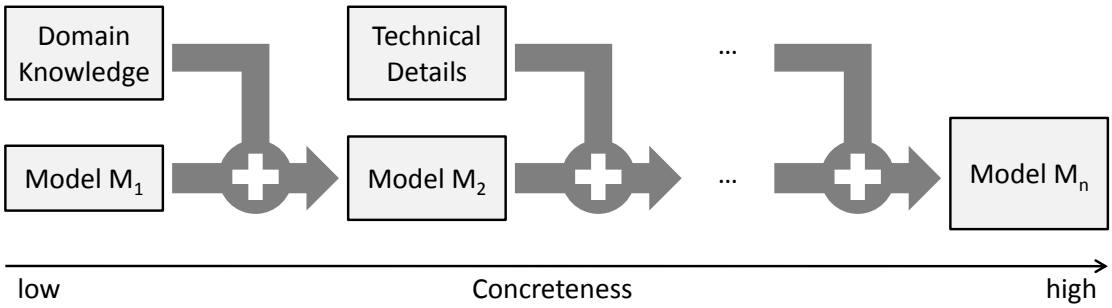


Figure 6.2: Enriching models with domain knowledge and technical details.

Starting from an abstract model M_1, more and more information is added by the use of model transformations. This information includes both domain knowledge and technical details, which are captured in models on their own. One can observe that the transformation process has a tree structure. Each derived model can be seen as a parent node, which is connected to its source models—its child nodes. A target model can never contain more semantically relevant information than the set of its source models [191]. Thus, all information that contributes to the enrichment process is originally provided by a leaf node. Thus, we can pinpoint another observation that motivates our approach:

O6-4 System models are assembled along chains of models. Each piece of information captured in a model, stems from one of its predecessors in the model chain.

If changes are applied to leaf nodes of the transformation tree, they can be propagated up in the tree by executing transformations. If changes are applied to intermediate nodes, they must be propagated to leaf nodes as these are the original sources of information. This particular observation (O6-4) does not necessarily apply to artifacts in arbitrary development processes, but in the case of MDSD it is very prominent. This is due to the compositional idea behind MDSD, which yields another important research question:

Q6-3 (Limited Generality) Can backpropagation-based RTE be applied to all RTE scenarios that rely on artifact composition?

Another fact that motivates the work in this chapter is the popularity of unidirectional transformation languages. For example, one can find many practical examples of ATL transformations whereas Query View Transformation Relational (QVTR) [83] is used less frequently. The reasons for this preference cannot be presented here in full detail, but we will try to collect some points in this regard.

First, the derivation of operational rules adds complexity for rule designers, because the gap between the execution and the specification is larger. Second, developers often start to write rules for the forward direction, where a unidirectional transformation

suffices. Third, bidirectional transformation tools (e.g., QVTR and TGG tools) derive a single source model when executing rules in backward direction. They are restricted to injective transformations. Fourth, tools for unidirectional transformations (e.g., ATL) were early available as part of mainstream development environments. Developers could easily use this new technology within existing processes. Tools for TGGs [97] did not gain the same degree of dissemination. Caused by all these factors, many existing model transformations are based on unidirectional approaches. Thus, extending such processes with RTE support and not requiring to rewrite existing rules fosters the acceptance of respective approaches in practice. We summarize this as observation O6-5.

O6-5 Many industrial model transformations are based on unidirectional languages.

In addition to the observations made w.r.t. practical solutions to RTE problems, we found some formal approaches [107, 156] that exposed a similar idea, but were not evaluated extensively yet. This gives rise to the next important research question:

Q6-4 (Feasibility) To what extent can theoretical work on change propagation be applied to practical RTE problems?

This chapter will address the resulting research questions identified above.

6.2 Overview and Definitions

To give an overview of our approach, we start with the following initial definition:

Definition 9: *A **basic backpropagation-based RTE system** is a tuple* (A, B, T, D, p) *where*

- A *and* B *are two domains (source and target),*
- $t : A \to B$ *is a transformation function from a function space* T*,*
- $D : \{d_1, \ldots, d_n\}$ *is a set of valid target changes* $d_i : B \to B$*,*
- $E : \{e_1, \ldots, e_m\}$ *is a set of valid source changes* $e_i : A \to A$*, and*
- $p : D \to E$ *is a propagation functional that maps target changes to source changes.*

Definition 9 formalizes the procedure shown in Fig. 6.1. The central idea is realized by the functional p, which translates target changes to source changes. By doing so, global consistency can be restored by the derivation of the new target model t''. The notion of consistency can be captured by a boolean function $consistent : (A \times B) \to \{true, false\}$ that takes a source model a and a target model b and returns $true$ if $t(a) = b$.

Definition 9 reveals that restoring global consistency is independent of a concrete p. No matter how target changes are translated to source changes, the replay of transformation t will ensure global consistency. Thus, the design of p is crucial to obtain a system that behaves reasonably. Users will expect that models—of both the source and target domain—are preserved to a certain degree. In particular applied changes must be preserved. To formalize "behaving reasonably", we define conservation functions.

Definition 10: *A **conservation function** conserved* $: (A \times A \times B \times B \times B) \rightarrow (0, 1)$ *maps the original source and the changed source model (of domain A), and the original target, the changed target and the echo model (of domain B) to a real number between* 0 *and* 1*. Higher values indicate that the changed source and the echo model conserve many properties of the original models and the changed target model.*

This function computes whether propagating a change preserves the involved models as desired (cf. Sect. 2.5). However, our *basic backpropagation-based RTE system* uses a functional p that derives exactly one source change for each applied target change. So, why should we evaluate the degree to which this source change conserves the involved models? Up to now, there is only one option—applying the change provided by p.

As we will observe later, practical RTE problems demand for propagation functionals that do not only yield one source change, but multiple. The reason for this is that a single target model change can often be triggered by a set of different source changes. To decide which source change is most appropriate, our approach requires the propagation functional to compute all valid source changes and then selects the most useful one using the conservation function. Therefore, we extend our basic definition as follows:

Definition 11: *A **general backpropagation-based RTE system** extends the basic backpropagation-based RTE system using a propagation functional* $p : D \rightarrow E_{set}$*, where* $E_{set} \subseteq E$*. Instead of mapping one target change to a single source change, the extended propagation functional maps target changes to sets of source changes.*

To translate changes from target to source domain, the relation between elements from both domains is essential. This relation is established by transforming a concrete model and captures the details of this transformation [66]. We define this relation as follows:

Definition 12: *A **traceability relation*** $TRACE_t \subseteq V_a \times V_b \times OP$ *relates elements* (V_a) *from model a with elements* (V_b) *from model b w.r.t. executing a model transformation t (cf. Def. 9) and a set of basic transformation operators* OP*. Elements of b are related to elements of a if they are derived using the respective operator from* OP*.*

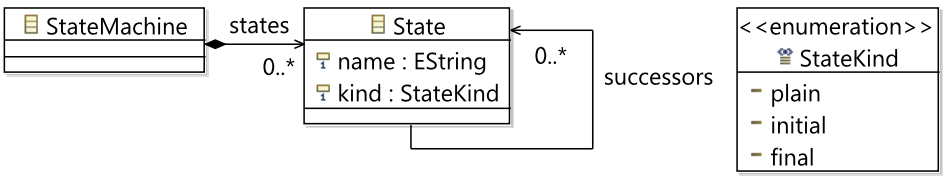


Figure 6.3: Minimalistic metamodel for state machines.

This definition for traceability relations establishes solely links between model elements, but not between references or attributes. Even though one could define similar relations for the latter kinds of elements, we restrict ourselves to links between model elements, because this is more close to practical transformations.

The four main components of the backpropagation-based RTE approach proposed in this chapter are therefore *change backpropagation*, *traceability*, *replay* and *conservation functions*. The first two components are concerned with the translation of individual target model changes to changes that need to be applied to source models. The third one computes the implications of the source changes to the target models. The fourth component decides which propagations are valid and which are more suitable than others. To clarify the roles of these components, the subsequent sections will present details of each of the four components.

6.3 Running Example

To illustrate our backpropagation-based approach to RTE, we will use the composition of state machines as an example. The language used to model these state machines is a simplified version of the state machine part of UML. Its metamodel is shown in Fig. 6.3. We use this simplified language, because UML uses classes to model transitions in state machines rather than references, which could cause unnecessary confusion in our subsequent explanations.

We will look at a transformation that is able to refine state machines by composition. The example is implemented using the Reuseware Composition Framework [188] which is founded on the concepts of ISC. For further details on Reuseware please refer to [192].

Since Reuseware supports the development of arbitrary composition systems, a specification of our composition system for state machines is required. This specification can be found in Appendix A1. We defined a composition system that allows to add new sets of states to existing state machines. Thus, existing state machines can be reused and extended according to new requirements. In Fig. 6.4, an example composition is shown[1].

[1]For presentational reasons transitions have names in the figures even though this is not supported by our metamodel.

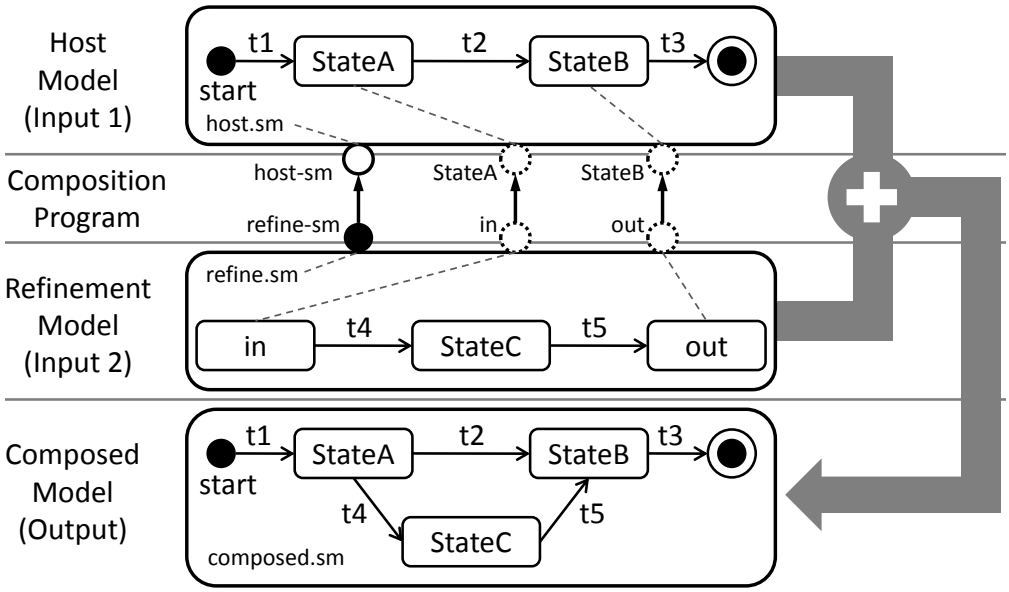


Figure 6.4: Example composition of state machines.

Here, two state machines (`host.sm` and `refine.sm`) are composed to yield a new state machine—`composed.sm` (shown at the bottom of Fig. 6.4). The instructions on how to compose the two state machines are captured by the composition program. In Fig. 6.4, the arrows that connect the `host.sm` and `refine.sm` boxes are a graphical representation of the composition program.

Simply put, `refine.sm` contains two states (i.e., `in` and `out`) that serve as placeholders. When `refine.sm` is embedded into another state machine, these two states are *merged* with two states of the embedding state machine. Thus, `in` and `out` cannot be found in the composed model. For the example at hand, `in` is merged with `StateA` and `out` is merged with `StateB`. This mapping is specified by the composition program.

The basic transformation operators involved in this example are `copy` and `merge`. The former copies elements from the input models to the target model. This is the case for all states except `in` and `out`. These are merged with `StateA` and `StateB` using the latter operator. `Merge` takes two arguments and has the semantics that all outgoing transitions (successors) of the merged element are attached to the target of the `merge`. The attribute values of merged elements are discarded.

The transformation that is realized by this composition system, takes a composition program, together with two or more state machines as input and produces a single composed state machine. Although, this is a quite simple transformation, the respective RTE scenario, exposes some interesting properties. For example, there are obviously multiple input models that yield to the same output model. The transition `t2` can be

part of `host.sm`, as shown in Fig. 6.4, but it could also stem from `refine.sm`. Thus, multiple source models are valid—a situation which motivates our approach.

6.4 Detailed Concepts and Process

6.4.1 Change Backpropagation

The previous sections introduced the notion of *change backpropagation*. Target changes are translated to corresponding source changes. This translation is realized by the propagation functional from Def. 11. To perform the adjustment of the source model, source elements that are affected by the change must be determined. Also, for a single target change, multiple changes in the source models may be required.

In this section, we present the details of this backpropagation of changes. We will first clarify the basic types of changes, then look at the translation of changes between domains and finally discuss the selection of the most suitable propagation option. The latter is needed, if multiple valid possibilities to propagate a change exist.

As we assume that all artifacts are instances of object-oriented metamodels, the elements that can be subject to change are objects, attributes and references. The basic changes that can be applied to these elements are:

- **Additions**—to create a new object,
- **Updates**—to change the value of an attribute,
- **Couplings**—to connect or disconnect objects with a reference, and
- **Deletions**—to remove an element.

For additions, we require a parent element, which can either be an arbitrary model element or the artificial root node mentioned in Sect. 2.1. By building sequences from these basic types of changes, models can be modified in arbitrary ways. For the time being, we will concentrate on the basic modifications and how to propagate them back. The composition of changes to form transactions will be discussed in Sect. 6.5.3.

To translate target changes to source changes, the traceability relation (Def. 12) is used. As explained in Sect. 6.3, our example composition system uses two basic operators—`copy` and `merge`. In the following, we will denote projections of the general traceability relation $TRACE_t$ for the respective operator as $copyof$ and $mergeof$ and we will also use infix notation. For example, $b_x \; copyof \; a_x$ is used to express that element b_x is the result of applying `copy` to element a_x. Also, the union of both relations $shadowof$ is used to denote that an element was either copied or merged.

A close look at the $copyof$ relation reveals that this relation is injective. For any target element there is exactly one source element[2]. However, this does not imply that

[2]Having at most one source element would also suffice.

Target Change	Valid Source Change(s)	Condition
Delete element b_x	Delete element a_x	b_x *copyof* a_x
Update element b_x	Update element a_x	b_x *copyof* a_x
Add element b_x to parent b_p	Add b_x to a_p	b_p *shadowof* a_p
Coupling element b_x to b_y	Couple element a_x to a_y	b_x *shadowof* a_x and b_y *shadowof* a_y
Decouple elements b_x and b_y	Decouple a_x and a_y	b_x *shadowof* a_x and b_y *shadowof* a_y

Table 6.1: Change translation strategy based on *copyof*, *mergeof*, and *shadowof* traceability relations.

the overall composition is injective too. As discussed before, this is rarely the case, because multiple source models can be composed to form the same target model.

In contrast, the *mergeof* relation is not injective. Multiple source elements can be merged into a single target element. Moreover, a single source element can be used multiple times in a composition, which implies that *mergeof* is not surjective either. Consequently, *shadowof*, being the union of *copyof* and *mergeof* cannot be injective. The properties of the three relations will explain some of the subsequent observations.

To understand the interaction between transformation operators, the traceability relation and change translation, consider our running example. The composed state machine contains copies of the models that served as input for the composition (cf. Fig. 6.4). For example, the states `start`, `StateA`, `StateB`, and `StateC` are copied to create the composed model. Also, the transitions `t1` to `t5` can be found in the composition result.

Table 6.1 shows an overview on how changes can be translated. First, if copied elements are deleted, the most natural way to propagate this change to source models is to delete the original, the copy was obtained from. Figure 6.5 shows such a modification. For example, if the final state is deleted from the composed state machine (1), it should be deleted from the respective input model. Second, performing updates on copied elements is similar. The respective original element needs to be updated. If `StateC` is renamed to `StateD` (2), the original of `StateC` must be renamed as well.

Third, additions made to the target model must be translated. For example, one can add a new state `StateE` to the composed model. In this case, the addition can trigger the creation of an equal state in either one of the source models, because the composed model was obtained by merging `host.sm` and `refine.sm`. Thus, it is not clear where the new state must be created. Both options are equally valid. We are facing a situation here where multiple valid propagation options exist. To decide which of the two propagations is the best, the synchronization fitness functions (cf. Sect. 6.4.3) are required. Without further knowledge about the intent of the developer who applied the change, the decision is not deterministic.

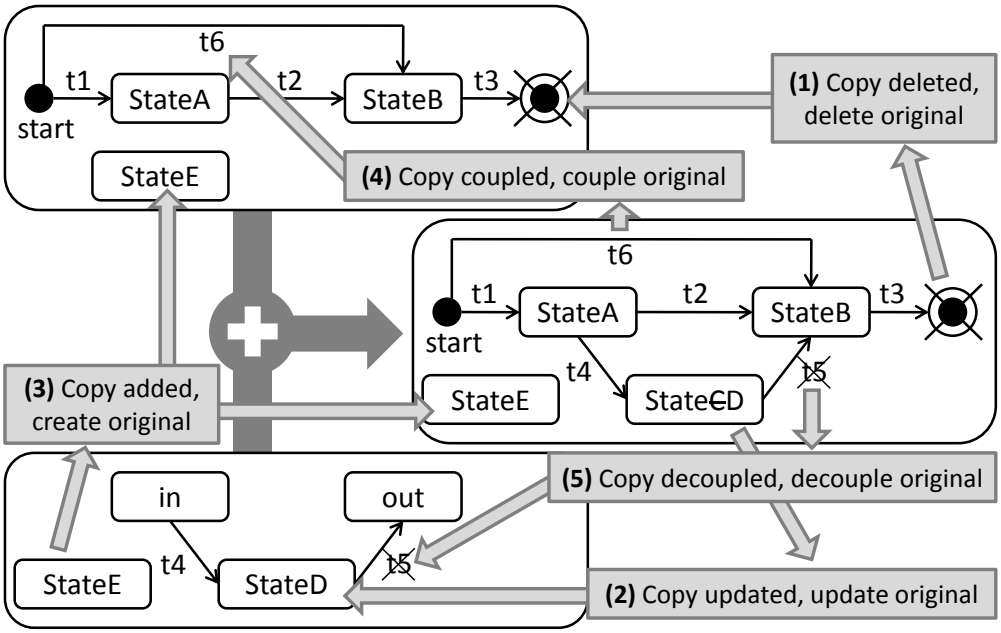


Figure 6.5: Propagating changes applied to a composed state machine.

Fourth, connecting elements with a reference (i.e., performing coupling) yields a similar situation. For example, if a transition `t6` (i.e., a reference) is introduced (4), the same reference must be added to the source model. However, if the reference is introduced between elements that result from merging multiple source elements, there are multiple options to create the reference.

Fifth, elements can be disconnected (5), similar to the deletion of model elements. To reflect such a deletion in the source model, the respective reference must be removed. In this case ambiguities can occur, if multiple references between two elements exist.

The examples show that some operations yield multiple options for translating a change. This is caused by the fact that the *mergeof* relation is not injective. Any change that is applied to a merged target element must therefore consider all elements involved in the respective merge. For example, adding a new state to the composed state machine can imply changing either one of the input models. The same applies to the modification of references. Only deletions and updates can be translated without ambiguity, because they solely rely on the *copyof* relation.

After examining the `copy` and `merge` operator, we can say, that the properties of these operators w.r.t. change propagation are quite profitable. Deletions and updates can be simply applied to the original elements. For additions and couplings, multiple valid propagation options can exist, but these are limited by the number of involved source

models. For other transformation operators, the derivation of source changes can be quite different and much more complex (cf. Sect. 6.5).

The concrete rules to transform target changes to source changes differ between RTE scenarios. They depend on the employed transformation technique. In particular, the semantics of the basic transformation operators matters in this context. Change translation rules can be reused, if the same operators are used. For example, our state machine composition relied on the `copy` and `merge` operators. Other transformation techniques, which do also copy or merge elements in the same way, may use the same rules.

From the example we make some general observations about propagating changes:

O6-6 Change propagation rules depend on the transformation operators in use, but not on concrete transformations.

O6-7 Traceability relations between source and target model elements are essential to propagate changes correctly.

O6-8 For a single change, multiple valid backpropagation options can exist.

The first two points imply main tasks when establishing RTE support according to our backpropagation-based approach for a specific scenario. First, one must either reuse or create a set of propagation rules that are feasible for the used transformation operators. This requires a thorough understanding of the operator semantics. Second, the relations between source and target models need to be known. Usually, this involves the implementation of tracing mechanisms to observe the execution of transformations to gain knowledge about the creation of target models.

The concrete complexity of these two tasks—specifying propagation rules and tracing transformations—depends on different factors. The propagation rules are inherently bound to the operator semantics and the effort to create them may therefore vary. For some transformation operators, reasonable rules might not even exist. In contrast, tracing transformations is merely a technical problem. Here, the cost of an adequate solution depends on the involved transformation tools and their support for traceability.

The last point—dealing with multiple valid backpropagation options—is particularly interesting. For example, if a target model element "depends" on two source elements, the deletion of the former can trigger the deletion of both source elements, or only one of them. If elements are added to a target model, there might be multiple possibilities for inserting respective source elements.

To deal with such situations, we propose a general procedure called *Propagate Replay Evaluate Pick (PREP)*. This procedure is also depicted in Fig. 6.6. First, we *propagate* all options in isolation, perform a *replay* (cf. Sect. 6.4.2) for each of the cases and then use the synchronization fitness functions (cf. Sect. 6.4.3) to *evaluate* the outcome of each propagation. Based on this evaluation, one of the propagation options is *picked*.

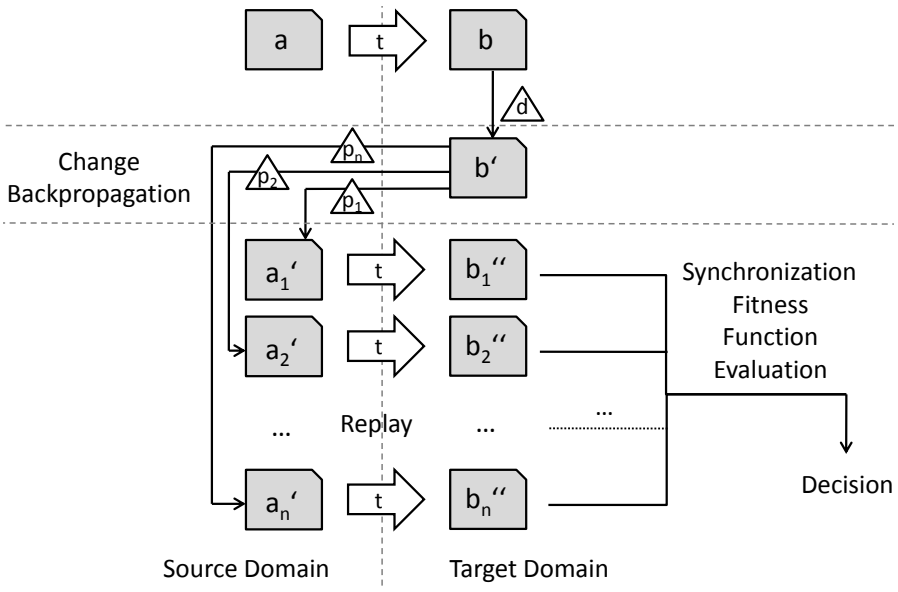


Figure 6.6: The Propagation Replay Evaluation Pick (PREP) procedure.

More precisely, a change d triggers multiple propagations (p_1,p_2,...,p_n). For each of this change sets a new source model (a'_1,a'_2,...,a'_n) is derived. By executing the transformation t on each of these source models, a set of target models (b''_1,b''_2,...,b''_n) is obtained. All these artifacts serve as input for the synchronization fitness functions, which evaluate the feasibility of each propagation (cf. Sect. 6.4.3 for more details on synchronization fitness functions).

The PREP procedure implements the idea of multiple backpropagation options. Here, some options may yield invalid source models (e.g., because they violate local constraints). Consequently, these options cannot result in a global consistent state. Moreover, some of the remaining options may be more feasible than others. To pick the option that reflects the expectations of the user best, synchronization fitness functions are used. Compared to approaches that use inverse transformations and which try to compute a set of synchronized models directly (cf. Sect. 6.5.1), PREP computes a variety of potential solutions and picks the best one.

6.4.2 Replaying Transformations

After computing a set of potential source change sets, individual versions of the source model—one for each potential change set—can be obtained. In Fig. 6.6, these are denoted as $a'_1, \ldots, a'_n$. Each variant reflects the situation that would be encountered if the respective change set $(p_1, \ldots, p_n)$ was going to be accepted. To view the impli-

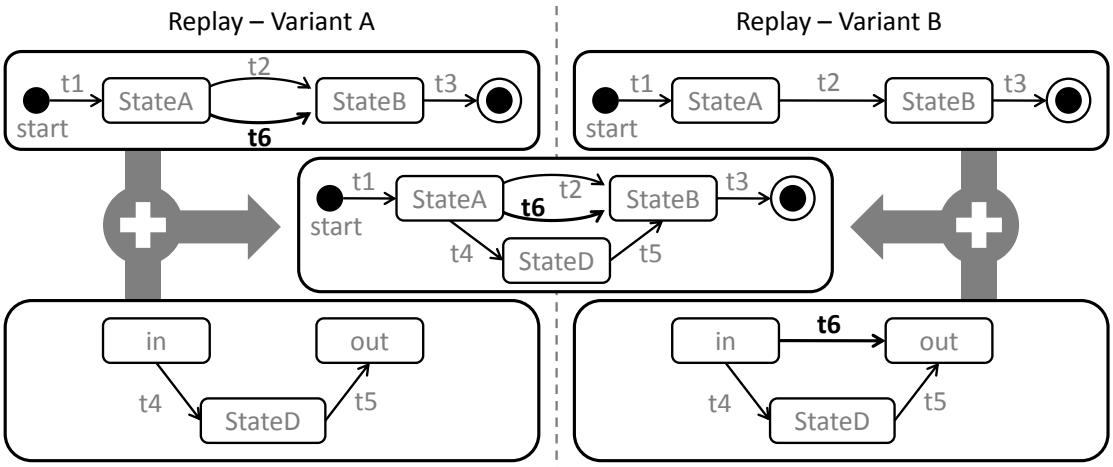


Figure 6.7: Replay of two different change propagations.

cations these change sets have on the target models, the transformation is executed using the modified source models as input. Consequently, each resulting target model $(b_1'', \ldots, b_n'')$ reflects the implications of propagating one change set. This process of executing the transformation on changed source models is called *replay*. The resulting models $(b_1'', \ldots, b_n'')$ are called *echo models*, because they reflect the impact of propagation choices by transforming the adjusted input models.

To explain the replay step using our example, consider the two compositions depicted in Fig. 6.7. Here, a new reference (transition `t6`) was added to the composed state machine (shown in the center of Fig. 6.7). This transition `t6` connects `StateA` and `StateB`. As mentioned before, there are multiple valid ways to propagate this coupling to one of the input models. First, `t6` can be propagated to the `host.sm` model—between the original states `StateA` and `StateB`. This variant is depicted on the left. Also, `t6` can be added between `in` and `out` in the `refine.sm` model, which is depicted on the right.

In this particular case, both propagations yield the same composition result when the transformation is replayed—b_1'' and b_2'' are equal. However, this is not generally true. For example, if a model fragment is copied multiple times within one composition, the result of the replay step can differ for varying change propagations.

The replay step is the most simple one as transformations do already exist and can be executed on arbitrary input models. The only technical difficulty that is connected with this step is to ensure that different versions of the source models are kept in isolation to make sure that replaying transformations does not interfere with other modeling processes. For example, transformations with static declaration of their input and output models need to be adjusted during replay to use the modified source model and to create a separate target model.

Besides the technical issues that need to be resolved, the replay step particularly contributes to the practical feasibility of PREP. The transformations are readily available and they can simply be executed on different versions of the source models. There is neither a need to rewrite the transformation rules using a different formalism, nor is a procedure required to automatically create inverses for given transformations.

6.4.3 Synchronization Fitness Functions

To decide whether models are in a synchronous state and whether this state is reasonably close to prior states, we employ the notion of *synchronization fitness functions*. There are two kinds of these functions—*consistency fitness functions* and *conservation fitness functions*. The former determine whether models are globally consistent. Thus, they are based on a particular state of a set of models. The latter compute the degree to which a state of a set of models is similar to a previous state of the same set. Conservation functions are therefore delta-based. Their input are sets of model changes.

Since the definition of the consistency fitness functions are defined upon tuples of models, we employ the following definition:

Definition 13: *A **model tuple** MT is a n-tuple $(m_1, \ldots, m_n)$, where m_1 to m_n are models.*

The first type of function (i.e., consistency fitness functions) is a slight refinement of Def. 1 and defined as follows:

Definition 14: *A **consistency fitness function** $consistent : MT \rightarrow \{true, false\}$ maps a model tuple MT to a boolean value ($true$ or $false$). If MT is consistent, the function returns $true$, otherwise $false$.*

In practice, it is usually sufficient to use a boolean function (i.e., a predicate) as consistency fitness function, because one is only interested in deciding whether models are fully consistent or not. However, there are authors proposing to allow for partial inconsistent models [5, 119]. For the scope of this thesis, we will not consider this case, but leave it to future work.

For the state machine composition example, the consistency fitness function can naturally be defined to return 1 exactly if recomposing all input models yields the output model. Otherwise, the function must return 0, because state machine models cannot be considered to be consistent if the composition yields a different result. Assuming that $in_1, \ldots, in_n$ denote our input models, *out* is the composed model and *comp* refers to the composition function, the consistency fitness function can be defined as follows:

$$consistent(in_1, \ldots, in_n, out) = \begin{cases} 1, & \text{if } comp(in_1, \ldots, in_n) = out \\ 0, & \text{otherwise} \end{cases}$$

This definition shows that the consistency fitness function does not incorporate the specific changes a user has applied to a composed model. The function is solely used to evaluate whether replaying a certain propagation option yields consistent models. It does not give information about which choice is more appropriate than others. This aspect is captured by another type of function, introduced in Sect. 6.2, Def. 10—conservation fitness functions.

In contrast to consistency fitness functions, boolean results are not practically feasible here. The purpose of computing the degree to which models are conserved, is to pick the most suitable (i.e., most conservative) backpropagation possibility. Thus, the conservation fitness function is used to find the most appropriate propagation option.

In the context of our running example, the conservation function can be designed to capture the number of changes that are applied both to source models—by propagating changes back—and to target models—by performing the replay step. Thus, the conservation function can be based on the number of differences between the original source models and the ones that are obtained by backpropagation of changes, as well as the differences between the changed composed model and the new composed model obtained after replay. Atomic differences do hereby correspond to the atomic types of changes introduced in Sect. 6.4.1.

The number of changes between two typed attributed graphs (i.e., two models) is often denoted as Graph Edit Distance (GED). Various algorithms exists to compute the GED (cf. [193] for an exhaustive survey). Even though computing the GED for arbitrary graphs can be difficult, we can employ specialized algorithms, as our models are defined by an EMOF metamodel (cf. [194, 195, 196, 197] for examples and [198] for a comprehensive comparison).

Since GED is an absolute measure, but the conservation function is required to be relative, a conversion is needed here. A pragmatic solution for this conversion is to compute the ratio between the number of differences and the size of the involved models. This reflects the fact that making many changes to a large model is equivalent to applying few changes to a small model.

Assuming there is a function *ged* computing the GED for two models, and another function *size* returning the number of elements found in a model, we can define a function $rdiff$ which computes the ratio between applied changes and the size of a model:

$$rdiff(m_1, m_2) = min(1, \frac{ged(m_1, m_2)}{size(m_1)})$$

If we assume that the number of differences is equal or greater than zero, and that the size of models is greater than zero (i.e., empty models are not allowed), $rdiff$ must

yield a value between 0 and 1. Based on $rdiff$, we can define $rsim$, which denotes the relative similarity between two models m_1 and m_2:

$$rsim(m_1, m_2) = 1 - rdiff(m_1, m_2)$$

A conservation fitness function for the state machine composition example can then be defined as follows:

$$conserved(i_1, \ldots, i_n, i'_1, \ldots, i'_n, o', o'') = rsim(o', o'') * \prod_{j=1}^{n} rsim(i_j, i'_j)$$

Here, i denotes input models and o denotes output models. Basically, the similarity between the original input models and their modified versions is computed and multiplied to obtain an overall similarity measure. Also, the similarity between the changed output model (i.e., the composed state machine) and the output model obtained after replay is taken into consideration by multiplying it as well.

The functions defined for this example are not fully determined by the RTE scenario itself. One can also use different functions depending on the concrete needs of users. We have presented very simplistic functions that try to make as few changes as possible to the involved models. We do not differentiate between different types of changes. Neither do we treat input models different from output models.

More complex conservation fitness functions can assign individual weights to different kinds of changes, for example, to prefer additions over deletions. Also, other conservation fitness functions can treat types of models differently. For example, modifications made to output models might be preferred over changing input models.

Depending on the concrete specification of the conservation fitness function, the outcome of the overall synchronization process can be controlled. The approach at hand is generic in the sense that it can be instantiated for different kinds of transformations and parameterized by a set of functions to steer the selection of the most suitable choice if multiple options for backpropagation exist. The design of the conservation fitness function is left to the developer of the RTE system.

6.5 Discussion

The previous sections introduced a novel approach to build RTE systems based on the idea of propagating changes that are applied to target models to respective source models. We have presented our approach using a simple transformation system that was able to compose state machines. Later on, in Sect. 9.1 more details on the RTE support for ISC will be presented. Also, the approach will be applied to template-based code generation in Sect. 9.2. Both sections will address Q6-3 (Limited Generality) and Q6-4 (Feasibility).

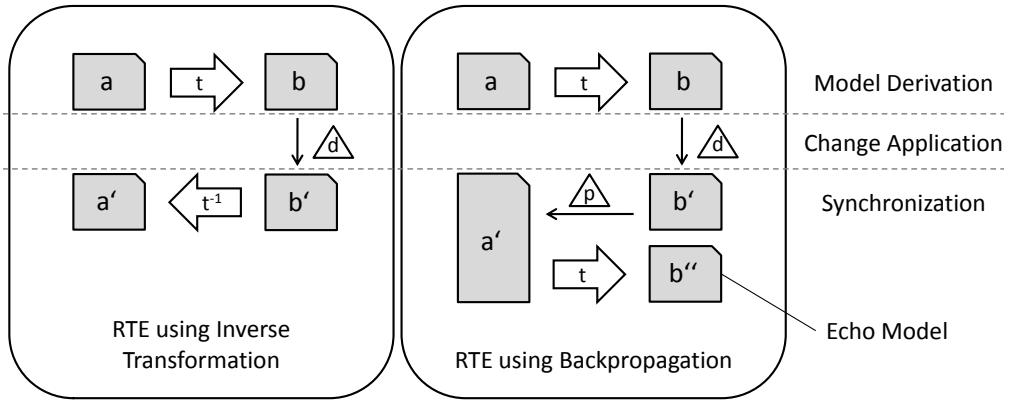


Figure 6.8: Inverse transformations vs. backpropagation.

To elaborate on the applicability of the approach at hand to other RTE scenarios, its limitations need to be carefully clarified (Q6-1 (Full Generality) and Q6-2 (Applicability)). Therefore, a couple of central questions need to be clarified:

- How is PREP different from using inverse transformations?
- Is backpropagation applicable to other transformation paradigms (e.g., if pattern matching is employed)?
- How can sequences of changes be handled?
- What if there are too many propagation options? Does the approach scale in this case?
- Which technical requirements are put forward by the approach?

The subsequent subsections, we will give answers to these questions and thereby try to point out the limits of our approach.

6.5.1 Comparison to Inverse Transformations

To reconcile source and target models in a RTE scenario, there are two basic alternatives. First, a transformation inverse to the original one can be employed. This case is depicted in Fig. 6.8 on the left. Whenever a change is applied to a target artifact, the inverse transformation is executed, yielding a new globally consistent source model. Alternatively, instead of using an inverse transformation, applied changes can be translated to a change in the source domain, as presented in this chapter. This alternative style of synchronization is depicted in Fig. 6.8 on the right.

Although, both types of RTE target the same goals, they are substantially different. Inverse transformations explicitly capture the information needed to transform target models back to the source domain. Whole target models are translated to obtain complete source models. In contrast, our approach translates individual changes made to a target model to changes that need to be applied to the respective source model.

To use the inverse transformation approach, the forward transformation must be known. If inverses can be derived automatically, a single RTE approach is sufficient to solve a certain class of problems. Namely, all RTE scenarios which involve the same class of (invertible) transformations can be tackled. But, if the derivation of inverses requires manual effort, no generic solution is available. The inverses must be created manually for each new transformation over and over again.

Bidirectional transformation languages (e.g., QVTR) avoid the problem of deriving inverses by construction. Instead of using two separate transformations t and t^{-1} (i.e., one per direction), a single specification is used for both directions. To do so, direction-specific transformation rules are derived. Even though the consistency of rules for opposite directions is preserved automatically, one cannot handle cases, where no source model a' exists, which strictly fulfills $t^{-1}(b') = a'$. In such cases, the execution of the bidirectional transformation cannot restore a global consistent state. Running a bidirectional transformation in target to source direction does simply not allow to alter the target model, which is inevitable here.

PREP does not require knowledge about concrete transformations. Rather, the basic transformation operators must be known to specify the propagation of changes. For example, if transformations copy elements from source to target models, the basic operator `copy` is used. For this operator, one can conclude that deletions of elements in the target model (i.e., copies of elements) imply the deletion of the original elements in the source model. No matter which concrete transformation uses the `copy` operator, the propagation of changes is the same. Our running example used this operator.

The contents of input models are copied to obtain the composed state machine. Moreover, not only the particular composition system for state machines, but all composition systems that implement ISC rely on the `copy` operator. Thus, a RTE approach that implements change propagation for this operator can be applied to all these composition systems, regardless of the concrete transformations they implement.

Furthermore, for backpropagation, the existence of an inverse transformation is not needed. To comprehend this, consider a transformation that transforms multiple types (e.g., `Keyboard`s and `Printer`s) to a single type (e.g., `Device`). That is, all kinds of devices in the source model are represented by a `Device` element in the target model, but cannot be distinguished anymore. There is no strict inverse available here, because information is discarded. Still, backpropagation allows to add new devices to the target model by reflecting such additions using a default device type in the source model.

The differences between RTE using inverse transformations and backpropagation are summarized in Table 6.2. As the backpropagation approach requires less details about

Aspect	Inverse Transformation Approach	Backpropagation Approach
Requirements	Transformation specification must be known	Transformation operators must be known
	Inverse transformation must exist	Inverse transformation is not needed
Granularity	Coarse	Fine

Table 6.2: Comparison of inverse transformations vs. backpropagation.

the RTE scenario, it seems more suitable to build generic solutions. In particular the fact that no inverse transformations are required, renders the approach attractive. Exactly the computation of such inverses is often difficult and sometimes not even possible [189].

6.5.2 Application to Other Transformation Paradigms

Different transformation paradigms rely on different transformation operators. When looking at such operators in the context of RTE, one must keep in mind that RTE is strongly connected to the relations between original and derived artifacts. Depending on the used operators, these relations can be quite different. The `copy` operator that is used in ISC establishes the *copyof* relation between model elements and allows for a simple translation of changes.

If we look at other kinds of transformations, in particular at heterogeneous ones, we can often find rule-based languages that rely on graph rewriting and thereby on pattern matching. In Sect. 2.4.3 various examples for such languages (e.g., ATL and QVT) can be found. These languages operate as follows. Whenever a certain structure is found in one of the source models, a corresponding, but different structure is created in a target model. The established relation can be denoted as *dependson*. This relation can be considered as a generalization of the *copyof* relation. Also, while *copyof* connects two elements only, *dependson* can hold between arbitrary many model elements.

The *dependson* relation connects one or more source elements with one or more target elements. Thus, changes cannot always be assigned to a single source element. Rather, changing multiple source elements can be required. Now, suppose we have a transformation system with the common pattern matching semantics (i.e., if a pattern is found in the input model, a corresponding model fragment is created in the output model).

The main difference between the `copy` operator and a pattern matching operator is that the former can be applied if the element to copy exists. The latter is applied only if multiple constraints (e.g., structural patterns) are met. The more complex these patterns are, the more constraints need to be checked before a pattern can be applied. Consequently, target elements depend on several constraints. This explains, for example, why deletions can be handled more easily in the context of the `copy` operator.

To propagate a deletion, only one source element must be deleted. In the case of pattern matching, multiple constraints were met to create a target element. Thus, each constraint is a candidate for invalidation. One can remove a single element of the ones that matched the pattern or change an attribute value that was required to fulfill the pattern's conditions. Also, one could remove all matched source elements or a subset thereof. All these changes can potentially remove the deleted target element after replay.

Similar arguments apply if target elements are updated or added. If the updated property was copied from a source element, backpropagation must update the source element. But, if the target property was computed using a non-injective function, multiple possibilities to change the source exist. For example, if the target property is the sum of two input properties, increasing the former by one can be realized by increasing one of the two input properties or any other change that yields the same sum. The conservation fitness function must then ensure that one of the first two options is picked. Nonetheless, computing all valid propagation options can be expensive.

So it seems, that the PREP approach is not particularly feasible for transformation systems that rely on pattern matching. Especially, large patterns and complex constraints imply an increased number of backpropagation options, which in turn require more computational resources. A closer look reveals the cause for the reciprocal relation between the complexity of patterns and the applicability of PREP.

Complex patterns are required if a transformation bridges a large gap between two domains (i.e., modeling languages). Concepts of one domain cannot be directly encoded using a concept of the other domain. Rather, complex patterns are required to express the same knowledge. As a consequence, the propagation of changes is rendered complex—PREP cannot be easily applied. If one considers the opposite situation where semantically close domains are bridged, patterns are more simple, often using one-to-one mappings. Naturally, the propagation of changes is more simple in this case.

After this discussion about the applicability of PREP to other transformation approaches that are based on graph rewriting, the generic use of PREP seems to be quite limited, in particular if complex patterns are required. However, one must keep in mind that some important RTE problems do not rely on pattern matching. First, we have already seen that PREP is applicable to the homogeneous composition of models. In Sect. 9.1, we will present an industrial case study for such a composition. Also, template-based code generation can benefit from PREP, which is a popular target for RTE. We will give details on this evaluation of PREP in Sect. 9.2. Thus, in summary one can say that PREP may not be particularly useful for transformations that employ pattern matching, but still provides a solution for other important RTE problems.

6.5.3 Handling Groups of Changes

Grouping changes can be required for different reasons. First, there are model editors (e.g., textual ones) where users do not apply individual atomic changes, but rather cre-

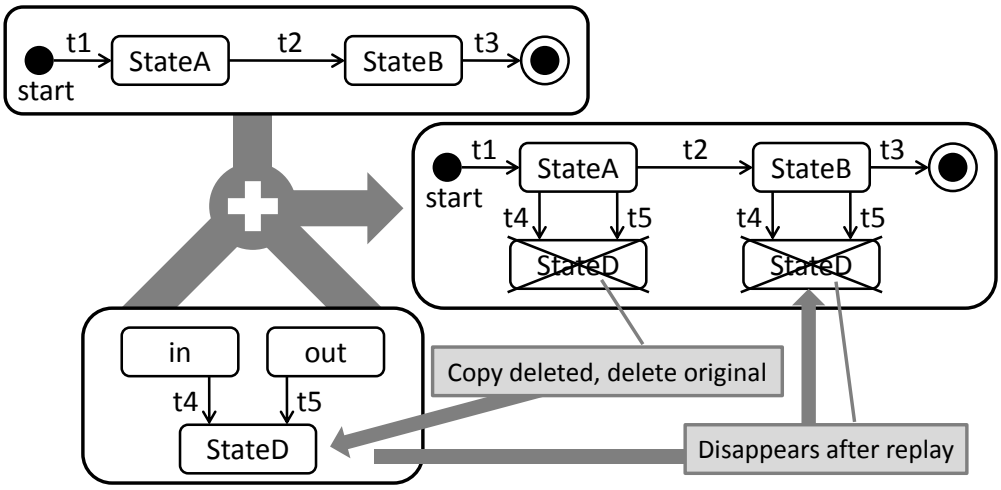


Figure 6.9: Conflict caused by grouping changes.

ate new versions of the model that is edited. Second, some RTE scenarios incorporate distribution, that is, models are edited concurrently by multiple parties. In all sketched scenarios, no information about individual changes is explicitly available. Rather, different versions of the same model are known and a difference algorithm must be employed to compute the differences between two versions [198].

Unfortunately, this procedure is not equal to recording changes while users edit models. First, the computed differences do not necessarily match the set of edit operations that were performed by the user. Users may have applied changes in a different order or may have chosen different kinds of operations yielding the same result. Algorithms that compute the differences between two models usually return the shortest sequence of edit operations. This is not necessarily the way users modify models.

The first issue is that backpropagation cannot be executed for each change. Consequently, the replay and evaluation steps cannot be performed as well. Therefore, target models are not updated right after each individual change. Rather, one must either execute the complete PREP procedure for each edit operation in the change sequence or propagate all edits to the source models first and then perform the steps replay, evaluation and pick. In both cases, the overall procedure can cause conflicts, if one of the edit steps within a group triggers a change in a target model that invalidates a subsequent edit operation in the same group. An example for such a situation is depicted in Fig. 6.9.

Here, the fragment depicted in the lower left of Fig. 6.9 is used twice in the composition and therefore two copies of `StateD` can be found in the composed model (depicted in the right part). If we consider a group of changes where both copies of `StateD` are deleted, an edit conflict can be observed. The deletion of the first copy of `StateD` would yield

the removal of the second copy if the deletion is propagated and replayed. But, since this is not done right away, the deletion of the second copy causes a conflict. The user removes an element which should not exist anymore. One can argue that the deletion of an element that was already deleted can simply be ignored, but the second edit operation could also have been changing the name of `StateD`, which can certainly not be ignored. Thus, solely grouping of edit operations can be problematic.

In other cases, the computed edit sequence may not exactly match the edits performed by the user. For example, a user may alter a target model by renaming an element `A` to `B` and adding a new element `A`. This is equivalent to a single edit (i.e., adding a new element `B`)—the result of editing the target model is the same. But, PREP can yield different synchronization results, because propagating the renaming of `A` can potentially impact many of its copies. If one solely propagates the addition of `B`, this is not reflected.

Even though we obtain a result that might not be expected by users, this is not a problem of PREP per se. If change information cannot be correctly provided by the editing infrastructure, there is less chance that PREP can compute the expected models.

To sum up, one can say that if no sufficient information about the model changes is available, applying PREP can be problematic. Such scenarios can be raised by specific model editors (e.g., textual ones) or distributed off-line modeling (i.e., when changes cannot be propagated immediately). The question how serious these restrictions are, can currently not be answered in full generality.

First, it is not clear how long sequences of changes can typically grow if editors cannot track individual changes. If sequences are short, PREP may still be applicable. Second, if PREP is employed in distributed modeling scenarios, the success of this does heavily rely on conflict resolution technology. Since we cannot avoid editing conflicts, the more important question is, whether they can easily be resolved by developers or even be handled automatically. The severity of all these restrictions does therefore heavily rely on the scenario at hand. During the evaluation of the PREP approach we will see that these concerns are often not as problematic as they seem at first.

6.5.4 Scalability

Another problematic concern when using the PREP approach are too many backpropagation possibilities. Mainly, because a great variety of potential ways to propagate a change requires more computations and thus more time and memory. Whether this additional computation effort is actually a problem, depends on the transformation system at hand. If individual transformations can be executed sufficiently fast, it may not be problematic to execute ten or hundred variants. But, one must keep in mind, that the PREP procedure is triggered after each individual change. Thus, it must be sufficiently fast to not disturb the editing of models.

The number of actual possibilities to propagate a change depends on the used transformation operators. For the ISC system, updates and deletions yield exactly one option

only. The addition of elements implied a set of solutions, because of the inherent indeterminism. However, the number of options was limited by the number of involved source models. For other transformation operators like the ones based on pattern matching mentioned earlier, the number of propagation options can be bigger. Here, one can experience an exponential growth of the number of possible propagations. The question whether the number of potential propagations is a threat to the application of PREP can only be answered by looking at concrete transformation systems.

6.5.5 Technical Requirements

The technical requirements connected to our approach, have already been briefly mentioned in Sect. 6.4.2. PREP requires an infrastructure that allows to *isolate models*. That is, one must be able to create independent versions of models to perform model transformations in isolation. Each propagation alternative yields a new version of the source models. Executing the transformation on these versions (i.e., performing the replay step) creates additional versions of the target models. All these versions need to be kept in isolation to avoid interference with other activities. Typically, this is not a very severe restriction, since versioning of models and executing the same transformation on different input models are typical requirements in MDSD.

In addition, if model editors do not support *change tracking*, an algorithm to compute differences between models is required. Since we operate on EMOF models, such algorithms are readily available (cf. [198]). If one wants to port the PREP process to other modeling paradigms, the same kind of differencing mechanism is required.

In addition to model isolation and change tracking, *establishing traceability* is required to employ PREP. One needs to have information about the concrete relations between source and target models available to provide accurate backpropagation options. For example, we keep track of all copies that are created from elements of source models in the state machine composition example. Without such fine-grained and precise tracing information, propagating changes becomes difficult or even impossible.

6.6 Summary

In this chapter, a generic approach to RTE was presented that can be applied uniformly to our example scenarios in Sect. 9.1 and 9.2. The approach is based on the idea that individual changes that are applied to target models can be propagated to source models. Thus, no inverse transformations are required, which makes the approach particularly applicable to scenarios where no such inverse transformations do exist.

Depending on the transformation operators that are employed in the forward transformation a specific relation between source and target model elements is established—the traceability relation. When changes need to be propagated the semantics of this relation determines whether and how target changes correspond to source changes. This is in

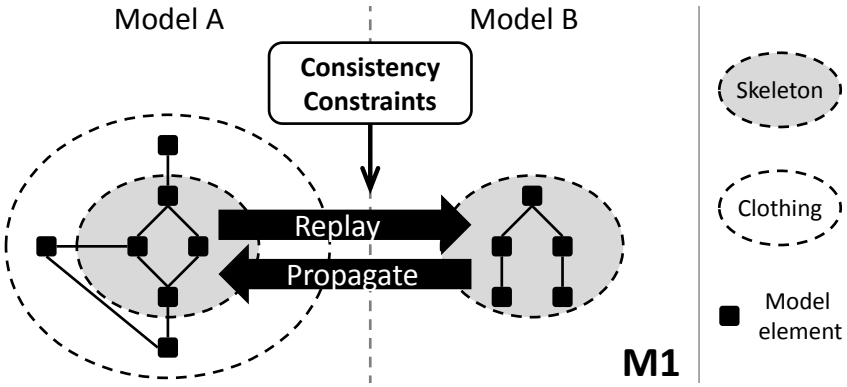


Figure 6.10: Model partitioning for backpropagation-based RTE.

contrast to [156], where traceability relations and basic transformation operators are not explicitly considered.

Sometimes, multiple valid options to propagate a change exist, which demands for a decision. To formalize this decision process, we have presented the PREP procedure that propagates each option individually and replays the transformation to obtain the resulting target models. Based on these target models, the usefulness of each option is evaluated using two functions—the consistency fitness function and the conservation fitness function. According to the design of both functions, PREP can be customized.

If we express PREP in terms of our conceptual framework, backpropagation-based RTE can be depicted as shown in Fig. 6.10. Here, the forward transformation is used to define when target models are consistent with source models. Since PREP requires that all target changes can be propagated to source models, the whole target model forms the skeleton. This is in contrast to the source model where parts can be irrelevant w.r.t. the forward transformation. These parts form the clothing of source models.

If target models shall contain information that cannot be propagated to source models, they need to be partitioned. Parts that are not reflected in source models (i.e., target model clothing) must be stored in separate model fragments and composed with the target skeleton to obtain complete target models. This composition is a separate RTE scenario. Our basic example—the composition of state machines—did not require additional partitioning. We will show more complex examples in the evaluation Sects. 9.1 and 9.2. For the former case study we will discuss such a partitioning of target models.

We have analyzed limitations and drawbacks of PREP and can clearly say that PREP is not a universal solution to all RTE scenarios. However, it allows for substantial improvement, at least for our set of scenarios. In addition, the support for non-invertible transformations renders PREP superior in comparison to approaches that target bijective functions. As such, PREP delivers a feasible approach for important scenarios in MDSD (e.g., model composition and template-based code generation).

7
Ontology-based Round-Trip Engineering

When models are synchronized manually, humans determine inconsistencies and resolve them step by step. To do so, they use special knowledge about the involved modeling languages. This knowledge includes the structure and semantics of the languages, constraints that might apply, and most importantly, the relationship between languages and their elements. Thus, to understand the process of model synchronization, it is crucial to grasp the relations between the involved modeling languages and the associated semantics. Moreover, relations within a single modeling language need to be considered when changes are made. Reestablishing a synchronous state among different related models can therefore be seen as propagating changes according to the semantics of all involved model relations—both within and across modeling languages.

Ontologies and DL, have been intensively studied as means to represent knowledge in a precise and formal way (cf. [55] for a comprehensive introduction). As such, ontologies seem a suitable candidate to formalize the manual synchronization process and thereby shift to a more automated synchronization approach. The challenge stated above (i.e., formally capturing model relations and their meaning) strongly suggests a closer investigation to clarify whether DL can be employed in this context.

This chapter is based on our paper "Ontology-based Model Synchronisation" [199] and further pursues this investigations. We will first motivate the use of ontologies and DL for RTE (Sect. 7.1). Secondly, we give an overview of our approach in Sect. 7.2, followed by a small example in Sect. 7.3, which will be used subsequently. Then, we will present how to specify inter-model relations using ontologies (Sect. 7.4) and how synchronization is performed (Sect. 7.5). We will carry out a discussion of the properties of the approach in Sect. 7.6 and conclude with a summary at the end of this chapter (Sect. 7.7).

Main contributions of this chapter are

C7-1 a novel approach to RTE based on semantically founded models,

C7-2 an analysis of ontologies (and OWL) w.r.t. their feasibility in creating synchronization specifications.

7.1 Motivation

When comparing the approach of knowledge representation taken in object-oriented modeling and ontologies, certain similarities can be observed. First, both share the concepts of classes and individuals (or objects, respectively). Here, the former describes a set of things, while the latter exactly refers to one of these things. Secondly, models and ontologies share the notion of generalization and specialization, which are subsumed by the term *inheritance*. Both representational styles use relations to describe semantic connections between concepts (or classes, respectively).

But, in contrast to models, ontologies (and DL being their foundation) provide a formal grounding that can be beneficial in the context of model synchronization. Ontologies allow to formally represent both knowledge about the subject of the synchronization and the rules that are used by humans during manual synchronization. The computational complexity of many of the underlying classes of logic is well understood. The derivation of implicit knowledge can be performed. All these properties make ontologies appear to be useful in the context of RTE, which motivated the subsequent investigations.

Moreover, various approaches to combine ontologies and metamodeling [13, 181, 200, 201] have emerged recently. These do explicitly target the formal description of model semantics. If such a description is available, the synchronization of models could be eased. In particular, there is hope that writing synchronization specifications could require less effort, if relations between metamodels can already be determined by the information about their semantics.

The previous chapter was motivated by the fact that some RTE scenarios require non-injective transformations and provides an alternative procedure to synchronize models. This chapter is focused on using ontology tools to synchronize models. Thus, it provides another alternative to classical bidirectional transformation languages. But, in contrast to our backpropagation-based approach, we do not aim to eliminate the need for inverse transformations, rather we want to investigate on the pros and cons of combining the formal grounding of ontologies with the practical feasibility of software models.

Within the PREP procedure, the synchronization fitness functions were designed to decide which set of models most appropriately reflects the change applied by a user while still preserving global consistency across the involved models. During this chapter, we will not see such functions, because the ontology-based approach to RTE uses predicates

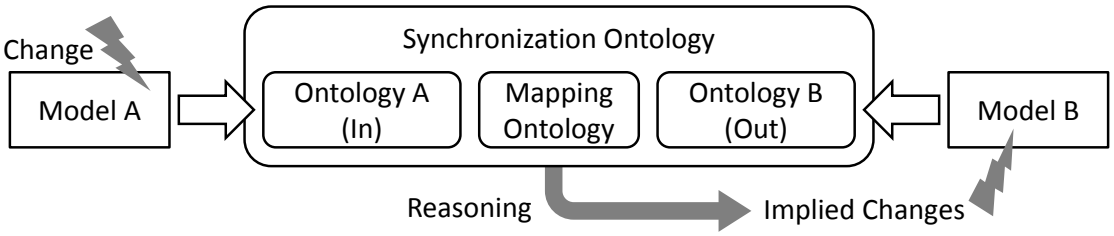


Figure 7.1: Schematic view of the synchronization procedure.

and logical rules to formalize how to synchronize models. In some sense, these predicates and rules are similar to the fitness functions, but the ontological synchronization approach does not employ a selection process. We do not explicitly compute sets of valid ontologies. This might be performed internally when employing a reasoner to process an ontology, but it is not an essential part of the approach presented in this chapter.

7.2 Overview

The basic idea behind our approach to ontology-based RTE is to translate models to ontologies and to employ ontology tools to compute the implications of changes. We perform the synchronization of artifacts in a different technical space.

To illustrate this using a simple example, consider a RTE scenario that involves two models. As we want to use ontological tool machinery for synchronization, both models and changes need to be transferred to a corresponding ontology, before the set of implied changes can be determined by reasoning. This procedure is depicted in Fig. 7.1.

To realize this procedure, three essential processes are required. First, models need to be translated to an ontological representation. This is required to let ontology tools process the information contained in object-oriented models. Second, additional knowledge must be provided to the ontology tools to configure the synchronization process. Without such knowledge, the relations between models are not known by ontology tools and meaningful results cannot be computed. Third, the output of the computations that are performed by ontology tools must be translated to respective model changes. After all, users must access the synchronization result in their model editors.

To switch between the modeling world and the ontology space, the mapping discussed in Sect. 5.4 is used. Thus, we can derive ontologies for EMOF models. We can also obtain models from a restricted subset of ontologies. This restriction is not a problem, since we do not create ontologies that cannot be translated back. Technically, the mapping is realized by the OWLizer tool, a part of the TwoUse toolkit [201].

Having the mapping between the two technical spaces available, we can focus on the actual synchronization of models. Here, the question is how to propagate changes

that are applied to one model to all related models. To do so, we must specify a set of consistency predicates and synchronization rules that can be evaluated when users apply changes to models. This information (i.e., how to propagate changes) is captured in a so-called *mapping ontology*, which corresponds to transformation specifications in the modeling world. This ontology refers to the involved metamodels by using the concepts that were derived from the metamodels. This mapping is handcrafted as it requires distinct knowledge of the involved domains (i.e., the respective modeling languages).

The fact that the mapping ontology is equivalent to the transformation specifications employed by model transformation languages, does also imply that the notion of a *synchronous state* is captured by this ontology. The models that are encoded in the ontologies are exactly in a synchronous state, if all predicates declared in the mapping ontology are satisfied (i.e., the whole synchronization ontology is consistent).

OWL-DL—being the concrete ontology language used in this thesis—is based on DL and is therefore strongly connected to set theory. OWL classes basically denote sets of individuals. Thus, the general idea behind our mapping specifications, is based on the consideration of classes as sets of individuals. One main objective of the mapping ontology is therefore to define the relations between these sets (cf. Sect. 7.4).

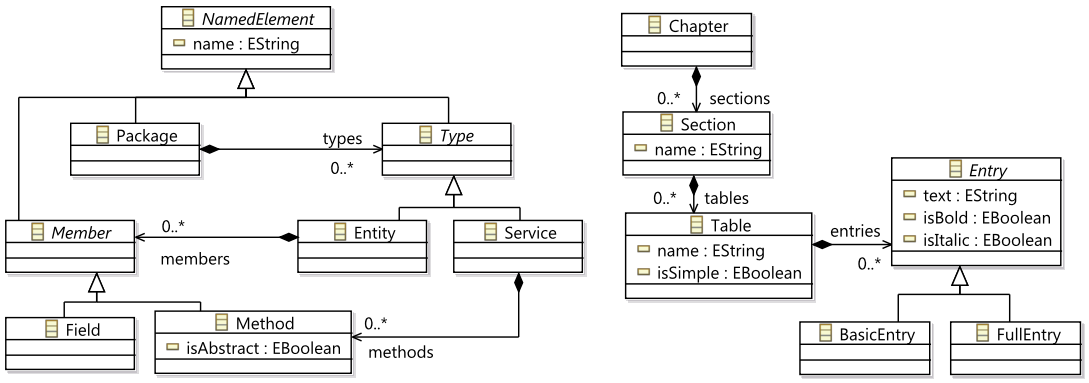
Evaluating the predicates and rules contained in the mapping ontology is performed in the ontology space. Thus, the obtained results must be translated back to the modeling world. For example, if the reasoning process determines that elements are missing, they must be added. Vice versa, if elements need to be deleted, we must remove them from the respective models. The concrete procedure to translate results of the reasoning process is quite comprehensive on its own. We will present its details in Sect. 7.5.

7.3 Running Example

To examine the specification of synchronization rules with ontological technologies in this chapter, we employ a simple example scenario. We consider the task of keeping entity models and their documentation in sync. The elements of the respective modeling languages are depicted in Fig. 7.2.

We can find `Package`s, `Entities`, `Service`s, `Field`s and `Method`s. `Service`s provide a set of `Method`s, while `Entities` can also contain data. `Field`s and `Method`s are concrete subclasses of the abstract type `Member`. Also, we can find the metamodel for the documentation, which consists of `Chapter`s, `Section`s, `Table`s, and `Entries`. `Entries` are similar to rows, but may actually contain more complex information. Also, there are `BasicEntries` and `FullEntries`. We consider the latter to contain more data.

The relation between the concepts of the two modeling languages is defined as follows. Each `Package` corresponds to a `Section` in the documentation. `Entities` and `Service`s are both represented by `Table`s. The latter use simple `Table`s, while the latter use complex ones. For `Method`s and `Field`s the mapping is slightly more complex. Abstract



(a) Metamodel for entity models.

(b) Metamodel for documentation models.

Figure 7.2: Metamodels for running example.

`Method`s and `Field`s correspond to a `BasicEntry` in a `Table`. All other `Method`s (i.e., all concrete ones) correspond to a `FullEntry`, which allows to add additional documentation (e.g., the results of a recent code coverage analysis). In addition, `Field`s are displayed in bold font, while abstract `Method`s are shown in italic font. `Chapter`s do not have an equivalent in the entity model.

The example is intentionally kept simple to ease the understanding of the subsequent sections. Although realistic synchronization tasks are much more complex, some of the basic patterns can be transferred from this example. We will show a more complex scenario during our evaluation in Sect. 9.3.

7.4 Specifying Model Mappings using Ontologies

The relations between elements (i.e., individuals) in models can be distinguished into different types depending on the number of related elements. For example, relations between individuals can be pairwise—one individual of one domain corresponds to exactly one in the other domain. An individual can also be related to a group of individuals in the opposite domain. In the most general case, groups of individuals are related to each other. These three relation types form the first column in Table 7.1. The first type (i.e., the 1:1 relation between individuals) can be further split into sub types depending on the number of involved classes. These types can be found in arbitrary RTE scenarios.

The most simple mapping can be found in row 1 of Table 7.1. Here, exactly one individual of the source domain corresponds to one individual of the target domain. In addition, within each domain, the individuals that are part of this mapping are instances of the same class—the two classes are equivalent. In our running example, such a relation can be found between `Package`s and `Section`s. Each `Package` corresponds to exactly

Individual Relation	Class Relation	Relation Name	Example Relation (Individuals)
1 : 1	1 : 1	renaming (class equivalence)	`Package`—`Section`
	1 : n	distribution	(`Entity`|`Service`)—`Table`
	n : m	cross-distribution	(`Field`|`Method`)—(`BasicEntry`|`FullEntry`)
1 : n	1 : n	unfolding	N/A
n : m	n : m	structural equivalence	N/A

Table 7.1: Types of mappings between domain models.

one `Section` in the documentation. The class `Package` is equivalent to class `Section`. We call such a mapping a *renaming* or *class equivalence*, since only the name that is used to denote the same set of individuals differs.

The next kind of mapping (Table 7.1, row 2) includes the case where single individuals correspond to a single individual in the opposite domain, but the latter may belong to different classes. Again, individuals form pairs, but the respective classes differ within one domain. Individuals that are subsumed by a single class in one domain are split into multiple classes in the other domain. Consequently, we call mappings with such characteristics *distributions*. In our example, this mapping can be found between `Entities`, `Service`s and `Table`s. Here, some `Table`s correspond to `Entities`, while others relate to `Service`s. The set of `Table`s is distributed across the types `Entity` and `Service`.

The most general relation that can be established by pairs of individuals is the *cross-distribution* (cf. row 3 in Table 7.1). Here, one individual still corresponds to exactly one individual in the opposite domain, but on the class level, multiple classes correspond to multiple other classes. Our running example does also expose such a mapping. `Field`s and `Method`s are both mapped to `BasicEntries` and `FullEntries`.

Even more complex mappings can be observed if more than two individuals are related, for example, if one individual corresponds to a set of individuals in the opposite domain. We call this particular case *unfolding* as the former individual must be decomposed to multiple individuals to represent it. Our running example does not expose this case, mostly because it cannot be easily handled with OWL constructs as we will see shortly.

The most general case of mapping holds between arbitrary many elements from both domains. Here, patterns of elements correspond to each other. We therefore denote this relation as *structural equivalence*. The example at hand does also not expose such a mapping, but one can certainly imagine how *unfolding* mappings can be extended to form structural equivalences.

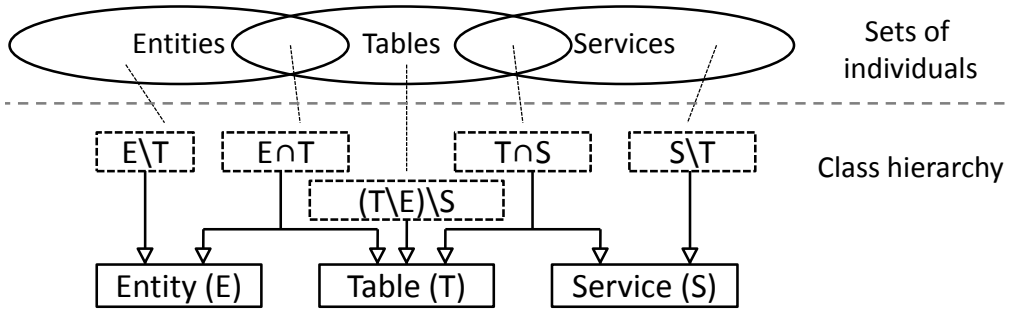


Figure 7.3: Example for mapping subsets of classes.

Having seen the different types of relations, the question is how to implement such mappings in terms of OWL constructs. We will present two approaches to do so. The first one (Sect. 7.4.1) is based on defining mappings using subclass definitions. The second one (Sect. 7.4.2) employs SWRL rules to define model relations.

7.4.1 Mappings based on Subclass Definitions

OWL classes and properties can be considered as sets. Each set contains all individuals that are instances of the class or property. When synchronizing models using OWL, a mapping is needed, which allows to determine all individuals that are required in the opposite domain to restore a synchronous state (cf. Sect. 7.2). To determine these individuals, the formalization of the relation between classes is needed. For the most basic mapping case (cf. row 1 in Table 7.1), classes can be defined as being equivalent. OWL explicitly provides the `EquivalentTo` construct for such purposes. In this case, subclasses are solely used to separate the mapping ontology and the domain ontologies. A respective class definition in OWL can be put as shown in Listing 7.1.

```
1 Prefix: owl: <http://www.w3.org/2002/07/owl#>
2 Ontology: <example://equivalence>
3
4 Class: Package          SubClassOf: owl:Thing
5 Class: Section          SubClassOf: owl:Thing
6 Class: Package2Section  EquivalentTo: Package, Section
```

Listing 7.1: Definition of equivalent classes in OWL.

The second row of Table 7.1, contains the first type of mapping which can be handled by defining subclasses—a distribution. First, consider the most simple case, where instances of one class (e.g., `Table`) are distributed over two classes in the opposite domain (e.g., `Entity` and `Service`). Instances of `Table` can either belong to `Entity` or to `Service`. In general, it is also possible that some instances of `Table` do not belong to any of the two classes. This situation is visualized by the Venn diagram in Fig. 7.3.

In the lower part of Fig. 7.3, the same facts are encoded in a class diagram. One can find the classes that are defined by the original metamodels of both domains—depicted as boxes with solid border. Then, for each potential case of mapping, a subclass is introduced. For a mapping between one class—in one of the domains—and two classes—in the other domain, we obtain five such subclasses. The number of these classes corresponds to the subsets that can be found in the Venn diagram above.

In Fig. 7.3, we consider the mapping between `Entities`, `Service`s and `Table`s. We introduce two subclasses of `Table`. One that denotes the set of `Table`s that corresponds to `Entities`, another one that represents all `Table`s that are related to `Service`s. These additionally defined subclasses use dashed borders. The first subclass (`E\T`) denotes all `Entities` that do not have a corresponding `Table`. In our example, this set is empty, but for other mappings it might be not. The second subclass (`E∩T`) represents all `Entities` that correspond to a `Table`. The definition of this class is crucial for the mapping. The concrete definition determines which `Entities` map to `Table`s and which do not. The third subclass (`(T\E)\S`) contains all `Table`s that are neither mapped to `Entities` nor to `Service`s. In our example this set is also empty, but this may not be the case for other mappings. The two remaining classes (`T∩S` and `S\T`) are reverse to `E∩T` and `E\T`.

By defining subclasses that extend classes from both domains, pairs of individuals can be represented as one individual belonging to two classes. For example, a `Entity-Table` pair is represented by an individual of class `E∩T`. If an `Entity` is created, one can reason that it also belongs to class `E∩T`, because `E\T` is empty. This implies that the newly created `Entity` individual is an instance of class `Table`. We will use this in Sect. 7.5.

We think this is a very intuitive way to create mappings. One must determine which sets of individuals in the source domain map to which sets in the target domain. Then, the sets need to be split until pairs of sets remain where individuals correspond pairwise.

This yields a general procedure of using subclass definitions. We can derive that arbitrary pairwise relations between domain concepts (i.e., metaclasses) can be handled in this fashion. Supposing that the individuals of the left domain are spread across m classes and the ones in the right domain span n classes, one can enumerate all intersections between the two sets of classes. In general the number of these intersection sets $count_{intersect}$ can be computed as follows:

$$count_{intersect} = m * n$$

For the basic example discussed above, this yields a number of 2 ($1 * 2$). Although we had five subsets in Fig. 7.3, three of them represented individuals that are not mapped to the opposite domain. Thus, no definitions (i.e., no subclass restrictions) are required for them.

If the classes from both domains that are involved in such a mapping are known, the intersection sets can be derived and the mapping ontology can be filled with respective subclass restrictions. For example, if a class extends class `A` and `B` (i.e., it is the intersec-

tion of two classes) we must specify which individuals belong to this class. Depending on this definition (i.e., the subclass restriction) a reasoner can decide which individuals of `A` correspond to individuals of `B` and vice versa.

We call the general process to create specifications for such pairwise correspondences *subset splitting and mapping*. It consists of the following steps:

1. determine all potential classes for left individuals,
2. determine all potential classes for right individuals,
3. derive set of intersection sets (i.e., subclasses),
4. add subclass restrictions to those intersection classes that are required (i.e., that actually contain individuals that are represented in both domains).

We split the classes for each ontology using subclass restrictions. By using the same subclasses (i.e., intersection classes), a mapping is established between the two ontologies.

The mapping based on subclass definitions does work for the first three types shown in Table 7.1 (i.e., all cases where pairs of individuals are related). It cannot be applied if multiple individuals take part in a relation, because domain classes are defined to be disjoint within one domain in the modeling space. Thus, we can only define non-empty subclasses that extend classes from different domains. Extending two classes from the same domain yields an empty set of individuals, because the classes are disjoint unless they inherit from each other (cf. Sect. 5.4.1). Besides this restriction, the approach seems very intuitive. Once the designer of a mapping has decided that individuals are related pairwise, she must clarify which set of individuals is mapped to which class in the opposite domain. The resulting set definitions can be formalized using OWL class definitions as shown in Listing 7.2.

```
1  Ontology: <example://subclasses>
2
3  Class: Entity   DisjointWith: Service
4  Class: Service  DisjointWith: Entity
5
6  Class: Table
7
8  DataProperty: isSimple  Domain: Table  Range: xsd:boolean  Characteristics: Functional
9
10 Class: Entity2Table   SubClassOf: Table  EquivalentTo: Entity, Table and isSimple value false
11 Class: Service2Table  SubClassOf: Table  EquivalentTo: Service, Table and isSimple value true
```

Listing 7.2: Subclass definitions for mapping Entities and Services to Tables.

Depending on the value of the data property `isSimple`, individuals of class `Table` are defined to correspond either to `Entities` or `Service`s. A reasoner can use these class definitions to determine the correspondences between the classes of the two domains.

We believe that splitting classes into subsets, where each subset corresponds to one class in the opposite domain, is easy to understand. Besides its simplicity, another advantage of the subclass mapping approach is that it is bidirectional. The mapping is independent of the synchronization direction. We can add individuals to any domain and infer missing ones in the opposite domain.

Moreover, the definition of subclass restrictions can be checked by OWL reasoners. Thus, one can easily detect overlapping subclasses, because these are reported by the reasoner as inconsistencies in the ontology. We have experienced this multiple times during our experiments.

7.4.2 Mappings based on SWRL Rules

A second approach to express mappings between domain models is based on SWRL. SWRL is an extension of OWL that enables the use of Horn-like rules in combination with OWL knowledge bases (cf. Sect. 2.3.3 for an overview of SWRL). SWRL rules represent implications and can therefore be read as follows. If all atoms in the antecedent are true, then the consequent must also be true. This is similar to model transformation languages that rely on pattern matching. If a specified pattern is found in an input model, a corresponding pattern is created in the output model.

But, in contrast to most unidirectional transformation languages that create a fresh target model for each execution of a transformation, SWRL rules can also be applied incrementally. That is, if a rule is already fulfilled, no new individuals are derived by the reasoning process. SWRL rules can therefore also be considered as incremental transformation rules.

The question that is instantly raised here, is how SWRL rules exactly relate to rule-based model transformation languages like ATL or Query View Transformation Operational (QVTO) [83]. Actually, both kinds of languages provide similar concepts to define rules. We can use constraints (i.e., predicates) to restrict the sets of model elements that match a rule. This includes type restrictions (e.g., specifying that some element must have a certain type) and data restrictions (e.g., to state that an element must have a certain name). Also, we can define structural patterns to specify the relations that must hold between elements in order to be eligible to match a rule, for example, to prevent the application of a rule if two elements are not connected by the correct reference.

One could say that SWRL rules share concepts with unidirectional transformation languages, with the exception that the semantics of SWRL is formally defined in [54] whereas the formal definition of the semantics for ATL and QVTO is still going on [91]. In Table 7.2 the concepts of SWRL are mapped to their intended purpose.

Thus, we can realize similar mappings as we could do with ATL or QVTO. We can specify rules that must hold between domains. The most simple rule—having two equivalent classes `Package` and `Section` (cf. row 1 in Table 7.1)—can be formalized using the following two SWRL rules:

$$Package(?x) \rightarrow Section(?x)$$
$$Section(?x) \rightarrow Package(?x)$$

Certainly, this is more verbose than using the `EquivalentTo` concept that is built into OWL, but it illustrates the idea. The antecedent of a SWRL rule is a pattern that must be matched in the source domain and the consequent is a pattern that must be present in the target domain. However, because we merged the source, the target and the mapping ontology into one large ontology—the synchronization ontology—we must ensure that SWRL rules do not mix domains.

SWRL rules can also be used to map references between domains. For example, the following rule maps the `entities` reference to the `tables` reference:

$$Package(?x) \wedge entities(?x, ?y) \wedge Entity(?y) \rightarrow$$
$$Section(?x) \wedge tables(?x, ?y) \wedge Table(?y)$$

Thus, we can express structural mappings (cf. row 2 and 3 in Table 7.1) using SWRL rules. However, the semantics of SWRL dictates that all variables in the consequent must be defined in the antecedent. This implies, that we can only realize mappings where the target structure has at most as many elements as the source structure. We cannot create new elements or references. This is a drawback compared to ATL or QVTO, where one can easily create more complex structures in the target model.

If we combine all previously explained steps to a complete process, we obtain the procedure depicted in Fig. 7.4. We start with an analysis of the relations that hold between the classes of our domains. For all relations where individuals correspond pairwise, we employ mappings that are based on subclass definitions (cf. Sect. 7.4.1). If relations are more complex, because individuals do not correspond pairwise, SWRL rules can be used to define synchronization rules. In this case, one must check that rules do not interfere. If rules do interfere, one must employ dedicated subclasses to resolve conflicts.

SWRL Concept	Purpose
DescriptionAtom	Definition of type constraints
DataRangeAtom	Definition of attribute range
DatavaluedPropertyAtom	Constraint on data property value
IndividualvaluedPropertyAtom	Constraint on object property value
SameAsAtom	Require equality
DifferentFromAtom	Require inequality
BuiltInAtom	Use primitive operation

Table 7.2: SWRL concepts and purpose.

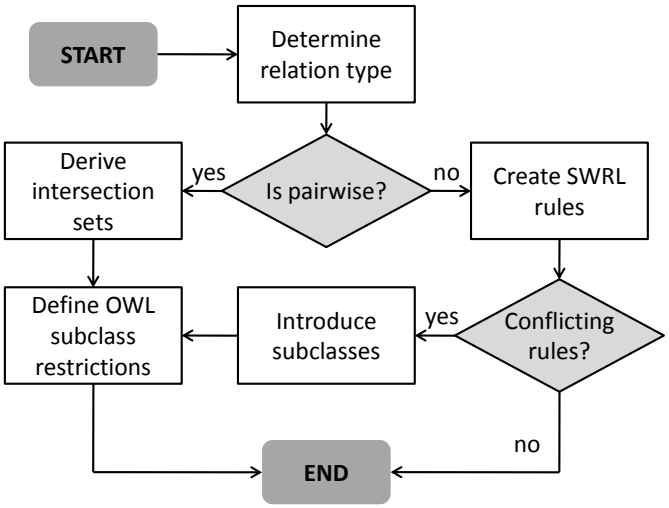


Figure 7.4: Process to create type and structural mappings.

7.4.3 Handling Primitive Data Types

In addition to classes and relations, metamodels can define attributes, which have a primitive type. These attributes are mapped to data properties while translating models to ontologies. When synchronizing models, we must consider attributes and their values. In principle, there are two cases that can appear when handling primitive types.

In the first case attributes are mapped directly. For example, the attribute `name` of a class in one metamodel is mapped to the identifier attribute of another metaclass. In this case no processing is required. To handle such mappings the previously described approach for synchronizing classes and object properties can be employed. Based on the definition of sub properties, the sets of attribute values can be distributed across the classes that need to hold these values. For example, the name of class `Table` can be split into two sub properties (i.e., the names for `Entities` and for `Service`s).

The second case can be observed whenever primitive values need to be processed (e.g., set conditionally or concatenated). In this case, we define new data properties and process these using SWRL built-ins. Such built-ins provide common operations of primitive data types. For example, the SWRL standard [54] defines built-ins for string operations (`stringConcat`, `substring` or `stringLength`).

SWRL built-in operations can perform complex computations. Thus, explicit specifications are required for each transformation direction. For example, if strings are concatenated by a rule, another rule is required to specify how strings are split.

To give an example, suppose we would like to prefix the names of all `Section`s that correspond to a `Package` with the string `pkg_`. To do so, the following rule can be added.

$$Package(?p) \wedge packageName(?p, ?pName) \wedge$$
$$stringConcat(?sName, "pkg\_"\hat{}\hat{}string, ?pName) \rightarrow$$
$$sectionName(?p, ?sName) \wedge Section(?p)$$

Here, the prefix and the value of data property `packageName` are concatenated. The result of the concatenation is set as value for property `sectionName`. To implement the opposite mapping a second rule is required:

$$Section(?s) \wedge sectionName(?s, ?sName) \wedge$$
$$stringLength(?length, "pkg\_"\hat{}\hat{}string) \wedge add(?total, ?length, 1) \wedge$$
$$substring(?pName, ?sName, ?total) \rightarrow$$
$$packageName(?s, ?pName) \wedge Package(?s)$$

Here, the prefix is removed from the name of the section. The remainder is set as value of data property `packageName`, which is the property that encapsulates the names `Package`s in the entity ontology.

Using SWRL built-ins we can process primitive values and thus incorporate them in synchronization specifications. However, one must be aware of the fact, that processing is restricted by the set of built-ins available in SWRL. One cannot perform arbitrary computations. In addition, we restricted ourselves to the DL-safe subset of SWRL, which implies that all built-ins that are not covered by this subset cannot be used either. For example, list operations are not DL-safe, which is why they cannot be used.

Having said that SWRL requires two rules—one for each direction—to process primitive types, we must emphasize a nice property of the formal foundation of OWL and SWRL here. During our experiments, we observed that reasoners were able to detect inconsistencies within rule sets. If pairs of rules did not match, an exception was reported. Thus, errors during the specification of synchronization rules can be avoided.

The general procedure to handle primitive types in ontology-based synchronization specifications is depicted in Fig. 7.5. First, one must decide whether values need to be processed or that they can be mapped unchanged. In the former case, SWRL built-ins can be used. In the latter, the definition of sub data properties is required. Finally, derived values need to be mapped appropriately to data properties of the opposite domain to allow reasoners to infer how values are propagated across domain boundaries. If the expressiveness of SWRL built-ins does not suffice, no mapping can be created. In this case, the restriction to DL-safe ontologies prevents the mapping at hand.

In summary, one can say that handling primitive types is not much different from traditional model synchronization approaches. Primitive types are processed using complex

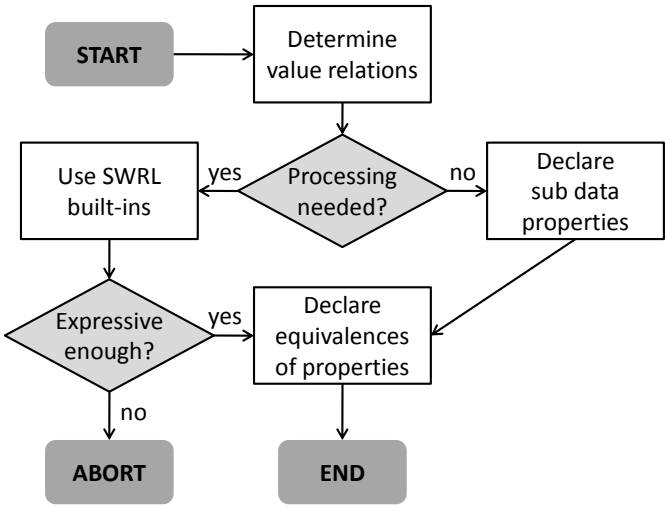


Figure 7.5: Process to handle primitive values.

and often non-invertible operations. This does leave no other choice than specifying individual rules for each transformation direction. Nonetheless, we believe that support for bidirectional built-ins in SWRL is beneficial in this context. Such built-ins are evaluated similar to relations (i.e., a reasoner must compute values for unbound parameters).

7.4.4 Reusing existing Semantic Specifications

Over the last years, a variety of approaches has been presented that aim at integrating modeling languages and ontologies [13, 181, 200, 201]. These approaches explicitly target the formal description of metamodel semantics. Their goal is to use metamodels not only to specify abstract syntax, but also to augment them with information that captures semantic aspects. For example, one can find proposals to integrate DSLs and OWL [200]. Also, an integration of UML and ontologies has been presented [201].

If metamodels are augmented with formal semantic specifications, the creation of synchronization rules may be facilitated. Up to now, all rules were created manually and the process was comparable to writing usual model transformation rules. We saw how to create mappings between types of different domains (Sect. 7.4.1), how to define structural mappings (Sect. 7.4.2) and how to handle primitive values (Sect. 7.4.3). But, none of these mappings made use of potentially available metamodel semantics. We worked with plain metamodels, metamodels that purely define abstract syntax.

In addition to approaches that integrate modeling and ontologies, previous work by Henrik Lochmann has shown, that multiple DSLs can be integrated on the basis of an ontological description [202]. He has shown that relations between domain concepts

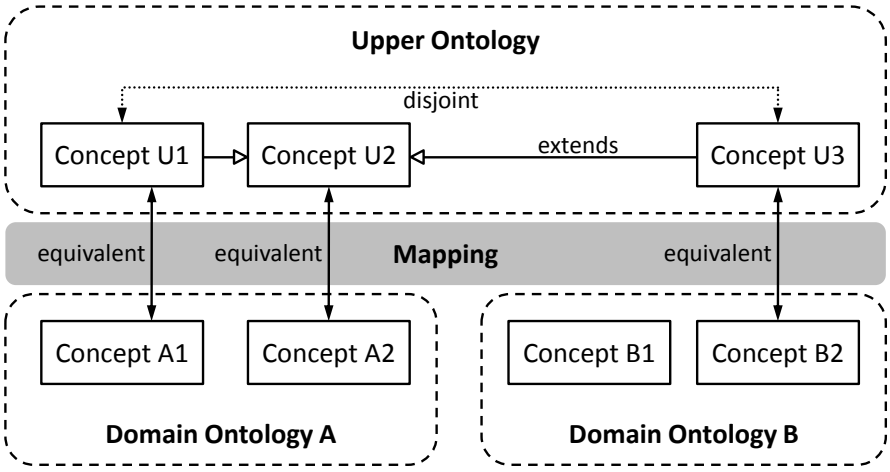


Figure 7.6: Mapping domain ontologies to an upper ontology.

can be captured by ontologies and that reasoning on these ontologies can foster the integration of artifacts written in different DSLs.

In [202] Lochmann proposed the use of an upper ontology—the Unified Software Modeling Ontology (USMO) [203]—to map the concepts of DSLs to a common semantic core. Similarly, an upper ontology can be employed to ease the specification of synchronization rules between different domains. Instead of creating a direct mapping between concepts of different domains, the concepts of each domain are mapped to an upper ontology. The general idea how to perform this is depicted in Fig. 7.6.

Figure 7.6 shows a concept `A1` that is declared to be equivalent to `U1`. The same relation does hold between `A2` and `U2`, as well as between `B2` and `U3`. Using the knowledge that is captured in the upper ontology we can already derive some information about the relation between domain A and domain B. For example, combining the fact that concept `U3` extends concept `U2` and the equivalences mentioned before, one can reason that instances of concept `B2` corresponds to instances of `A2`. The opposite is not necessarily true. Furthermore, the concepts `U1` and `U3` are declared to be disjoint. Taking this into consideration together with the equivalences that hold for `A1` and `B2`, one can derive that instances of `A1` cannot correspond to instances of `B2`.

Establishing such an indirect mapping using an upper ontology can therefore reduce the effort to map domain `A` to domain `B`. Of course, this assumes that a mapping to the upper ontology is readily available. If it is not, the specification effort can even exceed the direct mapping approach. But, if multiple domains are involved in a synchronization process, using a common semantic core—the upper ontology—can be beneficial.

To illustrate the mapping to an upper ontology further, we will return to our running example. If we recall the mapping from Sect. 7.3, we find concepts in both domains that

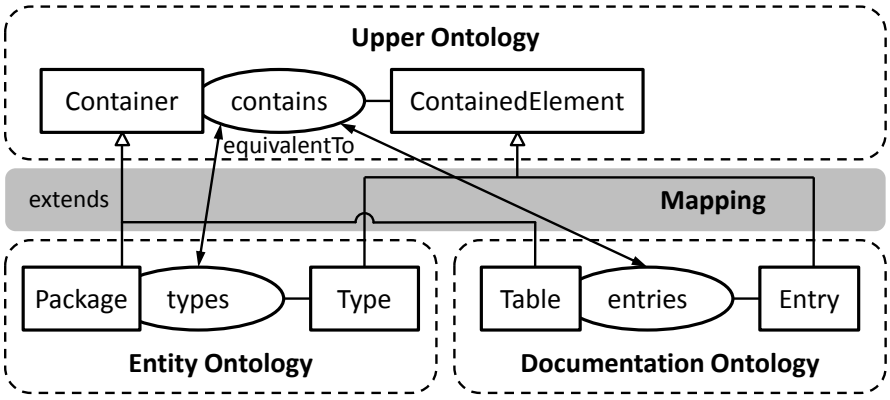


Figure 7.7: Mapping entity and documentation concepts to an upper ontology (excerpt).

are not only similar, but also quite generic. For example, entity models contain `Package`s and documentation models define `Section`s. Both classes serve as containers for other classes. The main difference is the type of the contained elements. `Package`s contain `Type`s, while `Section`s contain `Table`s. From an abstract point of view, both entity and documentation models provide two concepts that allow to store collections of elements. If we denote the elements that are collected as `ContainedElement` and the elements that collect as `Container`, we obtain the upper ontology that is shown in Fig. 7.7.

Here, the concepts `Type` and `Entry` are declared as extensions of `ContainedElement`. Similarly, `Container` is extended by `Package` and `Table`. Moreover, the object properties `contains`, `types` and `entries` are defined to be equivalent.

From this mapping we can derive that `Type`s contained in a `Package` can be considered as an `Entry`. Even though we did not create a direct mapping, the indirect mapping that is established by the upper ontology provides information that is useful to synchronize entity models and their documentation. Moreover, the two mappings—one for each domain ontology—can be reused if a third domain is mapped to the upper ontology. For example, if we require a tool to render documentation for the web, the modeling language that specifies valid input for this tool can be linked to the upper ontology.

7.5 Propagating Model Changes

In the previous section, we have presented different possibilities to specify synchronization rules using OWL and SWRL. To employ these rules in actual RTE scenarios, we propose the procedure depicted in Fig. 7.8. Here, the first step required to achieve synchronization is to capture model changes. This step is rather a technical issue and its concrete implementation depends on the editors used to modify models. In Fig. 7.8, this first step is depicted on the left.

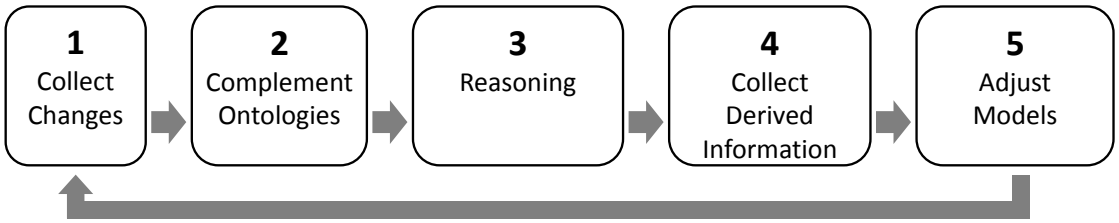


Figure 7.8: Change handling procedure - overview.

Once all changes are known, new information must be added to the domain ontologies. This constitutes step 2 in Fig. 7.8. Then, we use an OWL reasoner performing the realization task. More specifically, we use Pellet to compute a class tree, where each node contains all individuals that belong to the respective class. This tree is the outcome of step 3. To extract information that is of interest to us, we analyze this tree structure in step 4. Before the whole procedure starts over from the beginning, the models need to be updated to reflect the result of the synchronization in the respective editors.

The main steps in this procedure are step 3 and 4, where we employ reasoning facilities to derive actions that must be taken to restore global consistency. Step 2 is required to handle deletions, which requires some external logic for reasons we will present soon. All other steps (step 1 and 5) can be considered to integrate the models at hand with the ontological tool machinery.

To explain the details of the reasoning process, let us shortly recapitulate the main idea of our mappings. Both OWL subclass definitions and SWRL rules are based on the idea that corresponding elements are represented by a single individual. This individual has multiple types—one within each domain. Also, the types of the individual can be derived for both domains using our mapping specification. Thus, we can employ Pellet—or potentially another OWL reasoner—to infer this implicit information. To give a very basic example, let us consider Fig. 7.9.

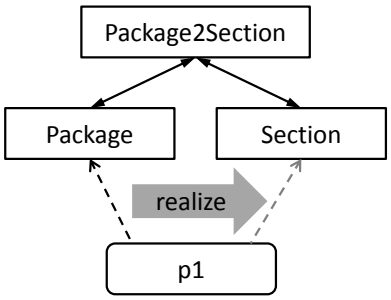


Figure 7.9: Determining type membership by performing realization.

Here, the mapping from Listing 7.1 is used to derive the type of individual `p1`, which is declared to be an instance of class `Package`. Using the specified class equivalences, a reasoner can derive that `p1` is also an instance of class `Package2Section`, and furthermore that it is an instance of class `Section`. Thus, the derived implicit information states that `p1` has both type `Package` and `Section`. Assuming `p1` was just added, we can imply that a corresponding `Section` must be created. If we let Pellet perform the *realization* task, we can calculate all types for all individuals, which is exactly what we require here. In a nutshell, additional types—besides the ones that are explicitly defined—are assigned to individuals, which allows to decide whether new model elements must be added. The implicit information is thereby made explicit.

Although this example is quite minimalistic, it illustrates the basic idea. We adjust the domain ontologies (step 2) according to applied changes (step 1), derive all information that is implicitly available (step 3) and adjust our models respectively (step 4 and 5).

Besides additions of model elements, the complete set of potential types of changes is the same as discussed for the PREP procedure in Sect. 6.4.1. We can add or remove elements, update attribute values, and couple (i.e., connect or disconnect) elements using references. Since we restrict ourselves to OWL-DL, which relies on monotonic logic, all changes that remove or change parts of models require special care. Monotonicity dictates that conclusions that have been drawn from a knowledge base must not be invalidated if new facts are added. One can easily imagine that removing an individual from an ontology invalidates previous conclusions, for example, the membership of the individual to domain classes.

Therefore, we divide the set of potential changes into *monotonic* and *non-monotonic changes*. Adding and connecting model elements belongs to the former category, while all other changes are subsumed by the latter. Both types of changes must be reflected by adjusting the domain ontologies. For all monotonic changes, we add new facts. For example, if a new object of class `Package` is created, a corresponding OWL individual must be added. Adding new references between objects is reflected by adding a new OWL object property assertion axiom.

All non-monotonic changes, require to adjust the axioms of the existing ontology. For example, changing an attribute value is reflected by adjusting the respective OWL data property assertion axiom. Deletions of both references and model elements imply the removal of the corresponding axioms. Non-monotonic changes are somewhat problematic to handle, because they invalidate facts that were previously derivable. For example, a model element b may have been added to the target model because a certain source element a existed. If a is deleted, the existence of b cannot be implied anymore.

We resolve this issue by keeping track of the relations between the elements and references that were created as a result of reasoning over the ontologies. This is very similar to the trace information that was employed in Chap. 6. We work around the monotonicity of DL by handling deletions and decouplings externally. To obtain a complete picture of our overall change handling procedure, consider Fig. 7.10.

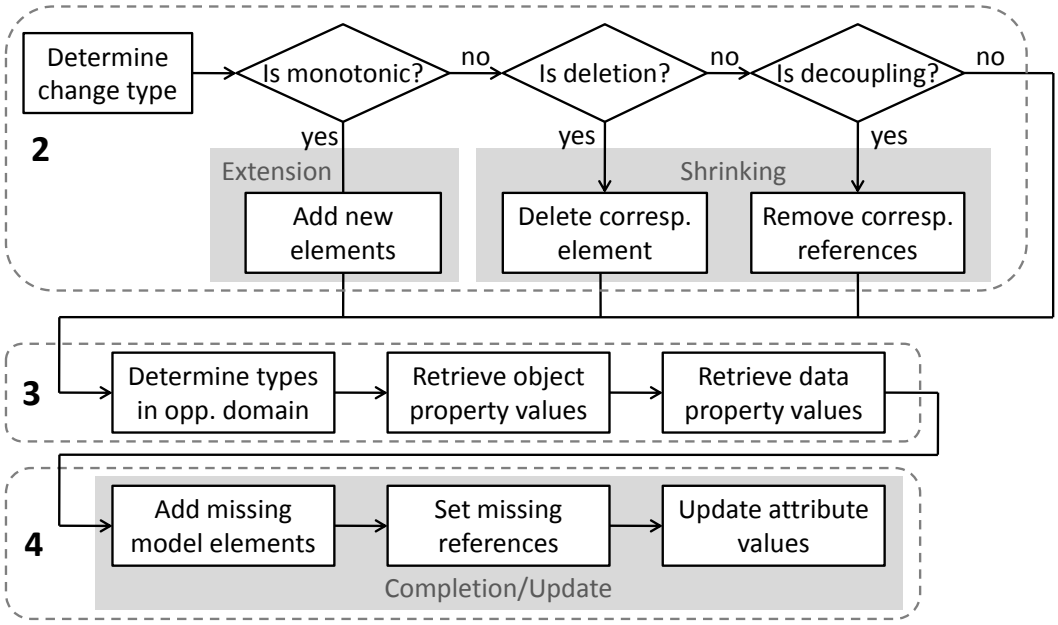


Figure 7.10: Detailed change handling procedure.

After determining the type of change at hand, all non-monotonic changes require to shrink the models (i.e., the domain ontologies). If elements are deleted, the corresponding elements must be removed as well. If a reference between elements is removed, all depending references (i.e., references, which were created as a consequence of adding the former reference) must be removed as well. We denote this part of the procedure as *shrinking*, because elements that depend on deleted elements are removed. The dependency relation is captured by the trace links that are captured previously. Updates do not need special treatment here, because attribute values cannot be created or deleted. They can be changed only.

Once the domain ontologies and the respective models have been reduced by removing all dependent elements, all implicit information can be derived by letting the reasoner perform the realization task. First, the derived types for newly added individuals are used to create corresponding model elements if required. We have seen a basic example for such a case in Fig. 7.9. Then, the values for all object and data properties are derived and the model elements are adjusted accordingly. If new object properties are found, new references are created. If data property values differ, the attributes of model elements are updated. This part of the procedure is called *completion/update*, because the models are filled and updated with the derived data.

We must note that the latter two steps—adjusting references and attribute values—require to traverse over the complete ontology. For each individual, we must retrieve

its data and object properties. This can be expensive if models and their respective ontologies grow large, but depending on the rules that are used for synchronization, arbitrary parts of the ontology can change. Unless there is support to incrementally reason over changing ontologies, this is inevitable.

7.6 Discussion

In this chapter, we presented a novel approach to establish support for RTE by using ontology tools. We have presented the approach using a simple example that involved entity and documentation models. To discuss the pros and cons of the approach, its limitations and differences need to be carefully clarified. The following questions provide a starting point for this discussion:

- How is ontology-based RTE different from using model transformations?
- Can ontology-based RTE be applied transparently (i.e., without requiring further knowledge about ontologies)?
- Can the approach handle large-scale applications?
- Which technical requirements are put forward by the approach?

In the next subsections, we answer these questions and sketch the limits of ontology-based model synchronization.

7.6.1 Comparison to Model Transformation Languages

In Sect. 7.4 we presented different possibilities to specify rules that capture the knowledge about how to synchronize domain models. We used subclass restrictions, equivalences, SWRL rules and upper ontologies to connect concepts of domains. From an abstract point of view, this was not much different from writing transformation rules in common model transformation languages like ATL or QVTO.

While we have used different syntax and also slightly different means to express relations between domains, in essence, we performed the same procedure. We identified and specified the conditions that must hold for one or more concepts of the source domain in order to be representable by another set of concepts of the target domain. The conditions were expressed using OWL subclass restrictions, SWRL atoms or by mapping to an upper ontology. In ATL and QVTO the conditions are expressed in OCL. Thus, a different constraint language is used, but we still define conditions that must be met by parts of models to be eligible for transformation, or synchronization respectively.

Now, what benefits can be gained from using OWL and SWRL? First, ontology-based synchronization relies on a formal semantic foundation. Therefore, the execution of

the synchronization is exactly determined and no space for semantic variations is left to the implementation of the transformation. In some cases, this can be considered a rather theoretical issue, but, if guarantees about the outcome of the synchronization are required, this property can become essential.

Second, inconsistencies between mappings and domain ontologies can be detected. We have experienced this while defining SWRL rules, where pairs of rules were required—one for each transformation direction. Checking these pairs for consistency comes for free, as we use the same formalism to express the data that is synchronized and the rules to do so. The same consistency checks for rules can be implemented for other model transformations techniques, but without a formal grounding, ensuring completeness and correctness of these checks is left to the developer of the transformation engine.

Third, creating indirect mappings between ontologies by using an upper ontology allows to reuse existing mappings (i.e., mappings that do already connect domain ontologies and the upper ontology). In principle, the same can be realized by transforming models to a common core. However, this requires to actually perform this transformation. That is, to transform a model from one domain to another, the intermediate model—the one that is an instance of the common core metamodel—must be created. If we employ an upper ontology and mappings from domain ontologies, we can perform the translation without actually creating the intermediate model. Rather, the mapping rules are combined to derive what the opposite domain model must look like.

Of course, there are not only benefits that can be gained when model synchronization is performed on the basis of ontologies. First, the specification of the mapping ontology requires knowledge about the involved languages (i.e., OWL and SWRL). Also, the designer of a mapping ontology must be aware of the translation from the modeling world to the ontological space. Since this mapping exposes some details that are not intuitive from a modelers perspective (e.g., features—object and data properties—are not attached to classes), one must be very careful to design a correct mapping.

Moreover, the mappings based on OWL and SWRL were restricted w.r.t. their expressiveness. Using these types of mappings one can only realize relations where individuals correspond 1-to-1. Complex, structural n-to-m mappings where $n \neq m$ were not possible. Using ATL or QVTO such mappings can easily be expressed.

7.6.2 Transparency of Ontology Tools

Initially, our goal was to hide the details of the ontology space. However, we quickly realized that this goal is difficult to reach. The amount of knowledge required to set up a mapping ontology and to test both its correctness and completeness, is enormous. While tools can certainly work in the background, configuring their correct input is impossible without detailed knowledge about OWL and the underlying semantics of DL.

Even, if one can come up with a set of consistent subclass restrictions and SWRL rules, debugging concrete synchronization steps is difficult. While reasoners like Pellet

can provide explanations for results of queries, connecting these explanations with the actual situation in the synchronization scenario is difficult. Too many translations occur, before model elements are actually synchronized. Determining the reason, why some change is not propagated as expected is therefore complicated.

7.6.3 Scalability

The scalability of the ontology-based approach to RTE must be evaluated from two perspectives. First, one must check how the specification of the synchronization rules (i.e., the mapping ontology) scales for large modeling languages. Second, one must clarify whether the actual synchronization process performs well for large models.

The first point—scalability w.r.t. designing the mapping ontology—can pose problems. Creating and maintaining a mapping ontology is quite difficult, because all concepts that can be used in an ontology are detached. For example, data and object properties are not explicitly connected to classes. This fosters their reuse, but also increases complexity. Even for the simple example of synchronizing entity models and their documentation, the resulting ontologies were difficult to maintain. This is also caused by the lack of sophisticated tools to edit ontologies. Ongoing work on sophisticated OWL editors[1] may provide solutions in this regard.

The second aspect—performing synchronization for large models—can also be problematic. Since reasoning tools are not specifically designed to perform model synchronization tasks, no optimizations w.r.t. time and memory consumption are available. In contrast, model transformation engines are explicitly built for this specific task. Thus, they can employ both optimized data structures and algorithms. Reasoners are built for a more general audience and can therefore not compete w.r.t. performance.

7.6.4 Technical Requirements

To apply our ontology-based synchronization approach, some technical requirements must be met. Most of them are shared with the PREP procedure (cf. Sect. 6.5.5).

First, model editors must provide information about changes that are applied to models. Whenever model elements are added, deleted or updated, we require notification. For some kinds of editors this is quite easy to establish. For example, for most graphical editors that are built on top of EMF, the EMF notification infrastructure is sufficient to observe changes. For other editors (e.g., textual ones) the procedure is more complicated. The reasons for this are along the discussion conducted earlier in Sect. 6.5.3.

Second, we require traceability, similar to the PREP approach. Without knowledge about the correspondences between individuals and references, deletions of elements and removals of links between them cannot be handled. However, keeping track of these correspondences can be realized easily.

[1] `http://www.emftext.org/language/owl2`

7.7 Summary

In this chapter, an approach to synchronize models using their ontological representations was presented. Based on existing work, models were mapped to ontologies and vice versa. To specify synchronization rules within the technical space of OWL, we presented several types of mappings that can be employed. One can either use subclass definitions to map model elements or SWRL rules to formalize relations between domain models.

The former idea is based on the notion of sets formed by individuals of classes. Subclasses are defined to divide sets and equivalence relations map subsets to corresponding sets of individuals in the opposite domain. We defined types of mappings that can be handled with this approach and other classes that cannot. Alternatively, one can employ SWRL rules, which sacrifices the bidirectionality that was available when using subclass definitions. We have shown a running example based on OWL and the Pellet reasoner to justify the approach practically.

The goal of this investigation was to *reduce specification effort*, *increase ease of specification* and *obtain bidirectionality*. Our results do confirm these partially. We found that mappings based on subclass definitions are intuitive, but unfortunately these can only handle pairwise mappings between individuals. Also, the need for bidirectional specifications was met by the subclass approach. SWRL, being a alternative specification language, does not possess this property. Whether specification effort is reduced, cannot be determined at this point. The technical obstacles one needs to deal with (e.g., when defining OWL classes instead of directly working with metamodels) are still to large to gain benefit from the formal semantic foundation. Modeling languages with built-in support for semantics [200] seem essential to overcome this barrier.

8

Role-based Round-Trip Engineering

In the two previous chapters, we have seen how RTE can be realized based on back-propagation and semantic technologies. These two approaches are particularly suitable if artifacts and tools do already exist and synchronization is required to ensure interoperability. However, instead of implementing support for RTE on top of an existing tool landscape, one can actively prepare tools to make them work together more easily. This can avoid many of the problems that have been tackled before, basically because future RTE scenarios are anticipated at tool design time. We have identified this need for anticipation in Sect. 3.3.

This chapter presents an approach that implements the idea of actively anticipating future RTE scenarios. It is based on the paper "Anticipating Unanticipated Tool Interoperability using Role Models" [204] and completes the initial ideas presented therein. We use role models to specify how tools access common data. Data that shall be used by multiple tools, is physically shared. Data is not replicated and synchronized, but rather adapted to specific representations required by different tools. The concepts to differentiate between shared and isolated data are defined on metalayer `M3`. Therefore, the approach is said to implement metatype-based partitioning.

Main contributions of this chapter are

C8-1 a novel approach to tool integration that uses role models to partition and share data across tools,

C8-2 an analysis of this approach w.r.t. its advantages and disadvantages over related methods.

This chapter is organized as follows. First, the motivation for using role modeling in the context of RTE will be given in Sect. 8.1. After a brief overview of how to use role models for tool integration (Sect. 8.2), a running example which will be used to explain the approach is presented in Sect. 8.3. The details of our approach are presented in Sect. 8.4. This will be followed by a discussion of the benefits and drawbacks in Sect. 8.5. Finally, a short summary is given in Sect. 8.6.

8.1 Motivation

A common situation in RTE is that different tools operate on specialized domain abstractions (i.e., separate metamodels). While this is reasonable from the perspective of tool developers, it poses problems when data shall be shared across different tools. For example, because of their different metamodels, tools may use different names for the same class of objects. Also, the functionality provided by tools is strictly bound to a metamodel. To apply tools to other data representations, one must employ transformations, which in turn introduce a great amount of redundancy (cf. Sect. 3.3).

Current object-oriented metamodeling languages like EMOF, allow to design metamodels for extension and interoperability. Various patterns exist, which allow the extension of metamodels [97]. However, tool developers are not enforced to use these. As a result, the decision whether to ease future interactions between tools or not is up to tool designers. Usually this yields limited anticipation for extension and interoperability, since tool developers have different objectives. Even sophisticated metamodel extension mechanisms introduce dependencies between tools [205]. One tool must extend the metamodel of another. If tools are developed independently, this is not feasible.

In an ideal world all future RTE scenarios would be known beforehand, which would substantially ease the task of anticipating every interaction between tools. However, new tools are created and existing ones evolve, which renders perfect anticipation impossible. One can also say that tools must anticipate the unanticipated.

An approach to handle unanticipated tool interoperability *a posteriori* is to apply transformations. Transformations convert data created by one tool to a different representation that is understood by another tool. However, in this case each tool keeps its own local version of the data, which yields to replication. This can in turn lead to inconsistencies. To resolve the implied synchronization problems, a posteriori RTE techniques, like the ones presented in Chap. 6 and 7 can be used.

Alternatively, instead of treating issues in RTE scenarios a posteriori, one can tackle the problems *a priori*. To do so, we propose to use role models. The concept of roles has been originally presented in [183] and was later on applied to object-oriented modeling [10]. In [10], a role model is defined as a unit to isolate an area of concern. This already indicates the parallelism to RTE, where different tools need to process shared data and each tool can be considered as a specific concern.

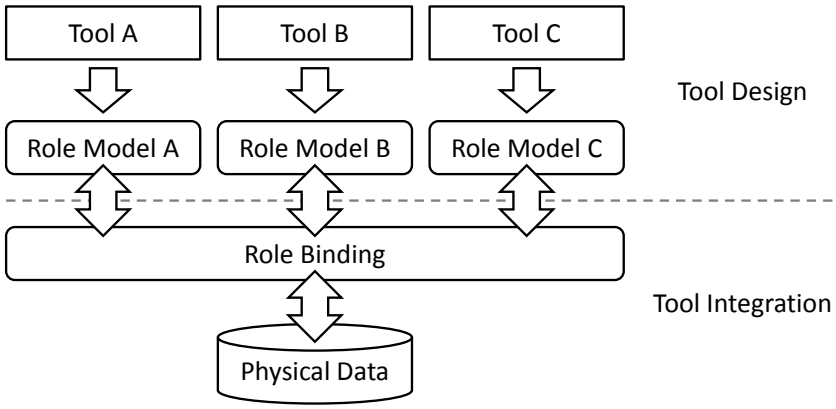


Figure 8.1: Role-based tool integration—Overview.

Role models capture properties that must be offered by a system to be used in a particular context. If we consider the application of a tool as a context, role models can abstract from concrete data repositories. Based on this abstraction, role bindings can be used to perform synchronization by connecting tools to a common data repository. The basic idea is therefore to share information physically, instead of synchronizing replicated data. Here, the question is, how much effort is needed to implement RTE-aware tool design. The objective of this chapter is therefore to investigate the promising advantages of role modeling.

8.2 Overview

The basic idea behind our role-based approach to RTE is quite simple. First, to actively anticipate future tool integration scenarios, we propose to use role models to specify the view on data required by tools. This view must capture all domain abstractions (i.e., concepts and relations) that are required by the respective tool.

Second, at tool integration time, these views are bound to other views or physical data representations. Decisions about interoperability, which were previously made by tool developers are left to tool integrators. This increases the degree of freedom at integration time and eases establishing support for RTE. To explain more details of this idea, consider the simplified overview of this approach that is depicted in Fig. 8.1.

Here, each tool accesses its data through a dedicated interface—its role model. The use of role models decouples the specification of the data required by a tool, from the binding of this interface to a concrete data source. Instead of letting tools access data directly, one enforces the usage of the role model. Tools cannot store and retrieve data in arbitrary ways anymore. The definition of this role model is performed at tool design

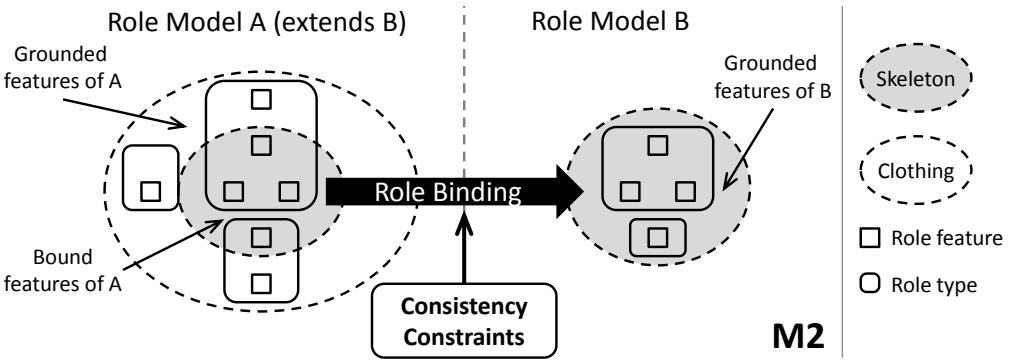


Figure 8.2: Role-based partitioning.

time. Thus, tool developers can still freely choose appropriate domain abstractions. Only the decision about how to physically store these abstractions is not up to the tool developer anymore. Rather, this task is assigned to the tool integrator.

At tool integration time, a decision is required how tools can actually access physical data. The individual role models need to be *bound* to concrete data repositories. To accomplish this task, a role binding must be specified. This binding specifies which data is retrieved from a physical location and which data is derived from other role models. In the latter case, role models are bound to each other.

In the simple case, where exactly one tool exists only, its role model must be bound to a physical representation as no other role models are available. However, if there are at least two tools, tool integrators can decide whether they want to bind role models to each other or not. In the former case—role models are bound to other role models—data is reused across tools. Different views map to the same physical data. This prevents redundancy and eliminates the need for tedious synchronization. In the latter case, each tools operates on its own physical data. Thus, data is not shared across tools. If we express this idea in terms of our conceptual framework, role-based RTE can be depicted as shown in Fig. 8.2.

Here, role models are used to define the partitioning of concrete models. The role features (i.e., attributes or references) that are bound to role model B, define which data is shared. These features form the skeleton, because they are kept consistent by referring to the same physical data. Role features which are not bound in role model A, form the clothing, because they are not shared with role model B. One can say that role-based RTE physically shares skeletons. The definition of shared data is performed on metalayer `M2` using concepts from `M3` (i.e., feature bindings and groundings).

To realize role-based RTE, some essential ingredients are required. First, a language to define role models (i.e., a metamodel for role models) is needed. Second, a language to bind role models to each other and to physical representations is required. We call this

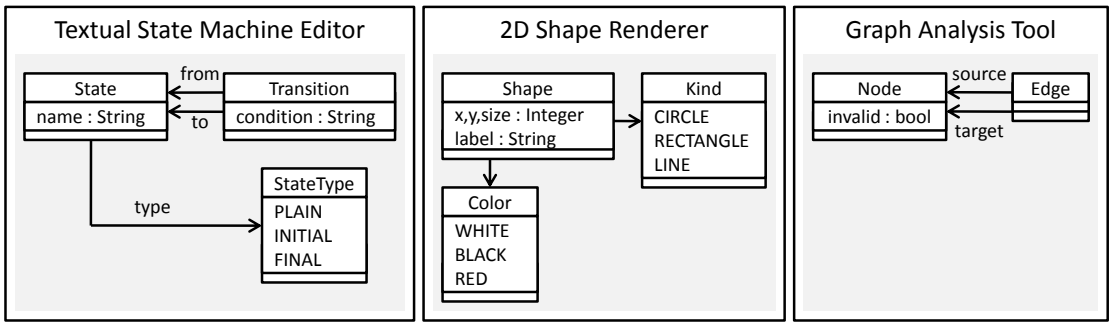


Figure 8.3: Example scenario.

language a *role binding language*. Supposing that both languages are available, we can handle future RTE scenarios by introducing new role bindings. Tool designers are then required to base their implementations on role models—tools need to be constructed differently. We will present details on both languages and role-based tool construction in Sect. 8.4. Also, we discuss the differences to RTE-unaware tool design. Here, the extent to which tool designers need to change their tool design process will clarified.

8.3 Running Example

To illustrate the role-based approach to RTE, we use a simplified, exemplary scenario of tools we would like to interact. Suppose, we like to create, visualize and validate state machines. To achieve this, we want to integrate three tools that employ custom tool-specific data abstractions. This scenario is depicted in Fig. 8.3.

First, we use a textual editor to create state machine models. This editor is aware of the domain concepts of state machines (i.e., states and transitions). We restrict ourselves here to this simplified representation of the state machine domain and do not incorporate concepts like guards, events and composite states.

Second, we like to visualize our textual state machines graphically, which is why a tool that can layout and render two-dimensional shapes is employed. We suppose that this tool was not specifically designed to render state machines—it rather uses the concepts of shapes, coordinates and colors.

Third, we want to statically check well-formedness rules for state machines. For this purpose, a generic graph analysis tool shall be used. Again, this tool is not aware of concepts specific to the state machine domain, but based on nodes and edges. Since the goal of the analysis tool is to find invalid elements, nodes carry an attribute `invalid`.

Using the graph analysis tool we want to check that initial states do not have incoming transitions and that final states do not have outgoing ones. We might also restrict the total number of transitions for one state to 10, because larger numbers indicate a bad

design of our state machine model (cf. [206] for more details about graph constraints and [207] for a similar example).

Each of the three tools owns a specific metamodel, capturing the domain abstraction that is appropriate for its task. The depicted metamodels are independent of each other. This allows tool developers to choose their domain abstractions freely. Also, changes made to one metamodel do not have implications to other metamodels. From the tool developers perspective, these are nice properties of the design depicted in Fig. 8.3.

However, our goal is to integrate the three tools such that RTE is enabled. We want to render state machines, which is why states and transitions need to be also usable as shapes. We would also like to apply the graph analysis tool to state machines. Therefore, states need to represent nodes and transitions need to be handled like edges. In other words, our tools need to share common data using different views.

Besides data that is shared across all tools (e.g., states and transitions), there is also data that is required by a subset of the tools only. For example, we might want to show errors stemming from the analysis of a state machine in red color when rendering state machines. Thus, the integration must allow the analysis tool to change the color of problematic elements.

The question raised by this example, is how the three tools can be built in order to allow RTE while preserving a high degree of independence. This RTE scenario includes the interactions mentioned above as well as other interactions, which were not anticipated at development time. Our goal is to illustrate how one can incorporate the RTE concern at tool development time and thereby ease unforeseen integrations in the future.

8.4 Integrating Tools using Role Models

To avoid data replication and the implied synchronization problems, we propose to use role models to capture the domain abstractions required by individual tools. Role models can be considered as an extension of object-oriented metamodeling. The concept of roles has been originally presented in [183] and applied to object-oriented modeling [10] (cf. Sect. 2.2 for an introduction to role modeling).

In [10], role models are defined as units to isolate an area of concern. This motivates their application to achieve tool independence. Each tool requires specific data structures on which it can perform computations. The definition of these structures is the concern that we want to capture by using role models. In addition to capturing domain concepts and relations (i.e., data structures), the employment of role modeling also introduces the notion of *role composition* [208]. This can be used to integrate several role models and object-oriented system specifications to an interacting system implementation.

In the following, we will elaborate how role modeling and role binding can help to achieve tool independence and interaction, respectively. We will first show how role models can capture domain abstractions for independent tools (Sect. 8.4.1). Then, col-

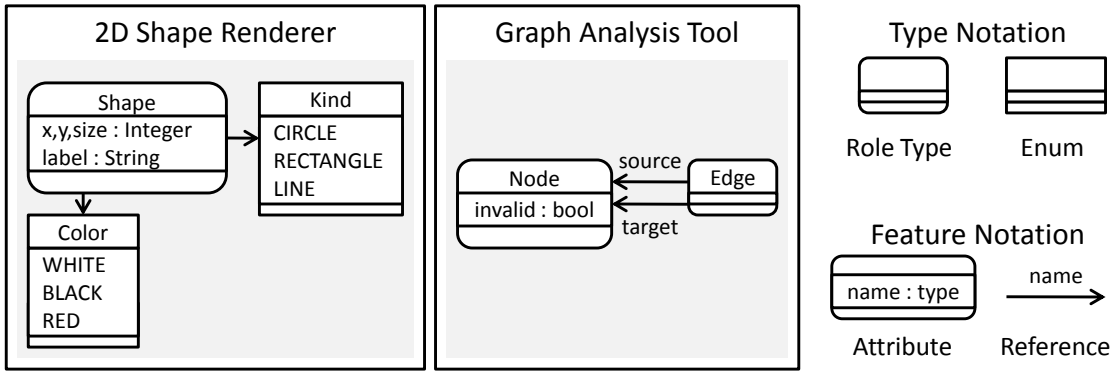


Figure 8.4: Example of role-based tool metamodels.

laborations between these independent role models are specified (Sect. 8.4.2). To make the approach practically usable, we will discuss how role models and role composition can be implemented in terms of plain object-oriented technology (Sect. 8.4.3).

8.4.1 Role Models for Tool Independence

In [10], a role model is defined to capture an *archetypical pattern of objects* that enables the specification of a specific concern of system behavior. Thus, role models can be used to define a specific data abstraction required by tools. Each individual role type defines a *type* of object required to achieve the functionality of a tool.

Figure 8.4 depicts a role-based version of two tool metamodels introduced in Sect. 8.3. Instead of concrete class-based data structures these metamodels use *role types* for all concepts of a tool metamodel. To model data structures required to implement tool functionality, role types can define *role attributes* that provide primitive values or *references* that connect several role types. Role models can, thus, be considered as a mechanism for defining the required data types of a specific tool. Compared to current object-oriented metamodeling approaches role models do not bind these data types to a concrete data source or representation.

As depicted in Fig. 8.4 for the 2D shape renderer, we use role attributes to specify that every player of the role `Shape` needs to provide a `label` and the `kind` of shape it is represented by. In the metamodel of the graph analysis tool role references are used to provide the `source` and `target` `Node` of an `Edge`.

A simplified metamodel of the language that is used to define role models is depicted in Fig. 8.5. It provides means to define `RoleModel`s consisting of several `RoleType`s. Each role type contributes a set of `RoleFeature`s that are either `Attribute`s with a `PrimitiveType` or `Reference`s with a complex `Type`. Also, `Enum`s are supported, which are considered as primitive data types.

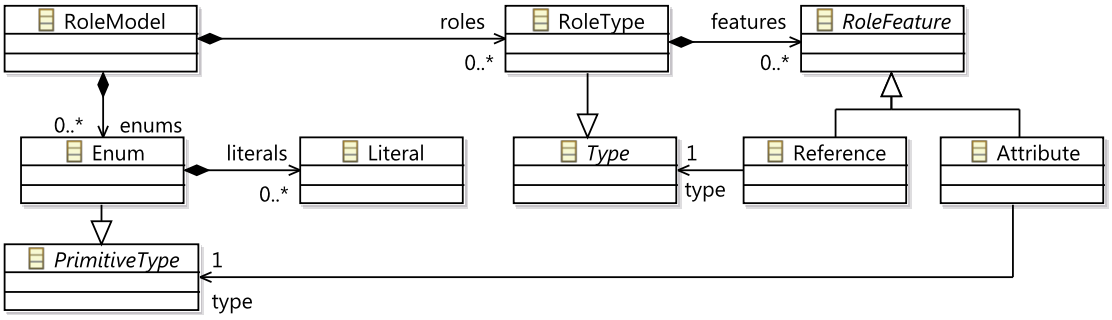


Figure 8.5: Language for role-based metamodeling.

The concepts that are provided by this role modeling language are similar to the ones of EMOF. For example, roles correspond to classes. Also, features both having primitive and complex types are provided by both. Moreover, enumerations can also be found in EMOF. This raises the question why a distinct conceptualization is needed here. Could we not just model the data types required by each tool on the basis of EMOF?

From a technical point of view, this is certainly possible and as we shall see in Sect. 8.4.3, one can employ EMOF models to realize the same objective. However, we have defined separate concepts instead of reusing the ones from EMOF to emphasize the different semantics. For example, a feature (i.e., an attribute or a reference) of an EMOF class entails the semantics that the value of the feature is physically stored within instances of the class. The value of an attribute is stored within the memory allocated for each object. In contrast, the concept `RoleFeature` shall put emphasis on the fact that the value of this feature can be retrieved from other sources.

Nonetheless, tool developers can use the language concepts depicted in Fig. 8.5. Role types can be used similarly to classes. Role attributes and references provide the same conceptualization as their EMOF equivalents. Using these basic building blocks tool designers can model the data structures required by their tool. This is not much different from creating a designated metamodel to capture the same knowledge. Basically, instead of defining sets of classes, role types and their respective features are specified.

8.4.2 Role Composition for Tool Interaction

After the specification of all role models, which is performed by tool designers, a procedure is required to integrate separate role models in order to obtain a coherent system, where tools can operate on the same data and thus all views are synchronized w.r.t. the underlying data. We achieve this integration by creating a *role composition model*. This model entails the relationship between role types of several role models. Role types of one role model are connected to the role types they play in another role model. Therefore, we also denote the composition model as *binding model*. To express such binding

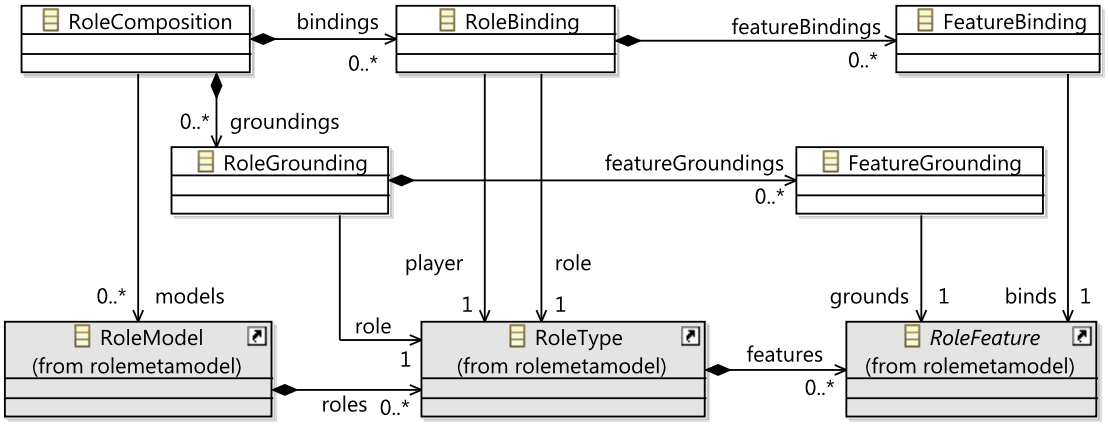


Figure 8.6: Composition language for role-based metamodeling.

models, we designed the Role Composition Language (RoCoLa). This language is based on previous work on language composition that was performed by Wende et al. [209]. The metamodel of RoCoLa is depicted in Fig. 8.6.

The concept `RoleComposition` is used to model the integration of multiple role models. It consists of `RoleBinding`s and `RoleGrounding`s. The former entail `FeatureBinding`s, while the latter aggregate `FeatureGrounding`s. First, we will look at `RoleBinding`s.

`RoleBinding`s connect two role types. One of the two role types is referenced to as being the player of this binding. As such, the player is entailed to provide all the features that are defined by the role types it plays. To specify how the player provides these features, that is, how their values are obtained, a set of `FeatureBinding`s is required. These bindings specify how a particular feature of a played role is provided. In the most basic case, a mapping to one of the features of the player is sufficient here.

Also, RoCoLa provides a concept to *ground* features of role types. Grounding a feature means, that the value of this feature is stored physically. That is, it is not derived from another role type. If one grounds all features of the involved role models, a system with isolated physical data repositories is obtained. We will explain the grounding of features in greater detail later on, but focus on role bindings first.

Examples for role feature bindings are shown in Fig. 8.7, where a graphical notation of RoCoLa is used. The two-headed arrows connect players with their played role types. Thus, these are instances of metaclass `RoleBinding`. To integrate the shape renderer and the state machine editor, both `Transition` and `State` must play the role `Shape`. Playing the role `Shape` requires respective players to provide all features defined for this role type. For example, every player must provide a label. Since both `Transition` and `State` do not directly provide a label, one must specify how the labels of transitions and states are obtained. This is not depicted in Fig. 8.7, but we will discuss this later on.

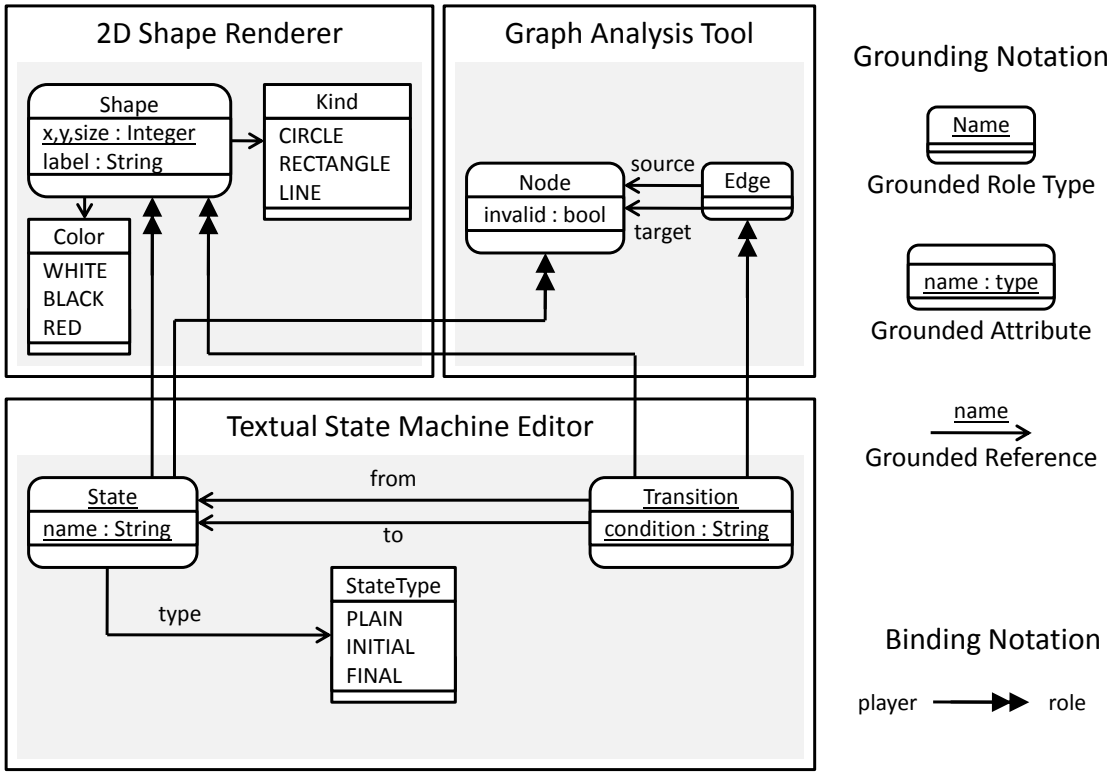


Figure 8.7: Example of role composition for tool interaction.

The binding of the role types `Transition` and `State` shows that role types can be bound by several players. Likewise role players can be bound to several role types. To integrate the metamodels of the state machine editor and the graph analysis tool, `Transition`s are bound to `Edge` and `State`s are bound to `Node`. Binding `Transition` to `Edge` requires transitions to provide a reference to a source and a target node.

We can say that the role composition depicted in Fig. 8.7 specifies a coarse grained binding of role types to other role types, but some details are missing. Most prominently, no information is available about how to obtain the bound features. To express this, RoCoLa provides further concepts. In addition to the concepts depicted in Fig. 8.6, the role composition shown in Listing 8.1 employs special feature binding types.

For example, RoCoLa provides a concept to map a feature of the player role to a feature of the played role. For example, the role type `State` playing the role type `Shape`, binds feature `label` to its own feature `name` (Line 3). In addition, Listing 8.1 employs a conditional operator to provide different feature values depending on the value of other features. For example, the value of feature `kind` is determined depending on the value of the `type` feature in role `State` (Line 4 and 5).

```
1  integrate statemachine, 2dShapes, graph {
2    State plays Shape {
3      label: name
4      kind:  if (player.type == PLAIN) return RECTANGLE
5             else return CIRCLE
6      color: if (player.type == INITIAL) return WHITE
7             else return BLACK
8    }
9
10   Transition plays Shape {
11     label: condition
12     kind:  return LINE
13     color: return BLACK
14   }
15
16   State plays Node {}
17   Transition plays Edge {
18     source: from
19     target: to
20   }
21
22   ground State { name, type }
23   ground Transition { condition, from, to }
24 }
```

Listing 8.1: Example role composition specification.

The language used here to specify these bindings is rather declarative, but one could also employ an imperative language (e.g., Java). Depending on the implementation of the role bindings (cf. Sect. 8.4.3), they must be translated to the target platform of the integration or interpret at runtime. In any case, the properties and expressiveness of the role binding language requires special attention in the context of RTE (cf. Sect. 8.5.2).

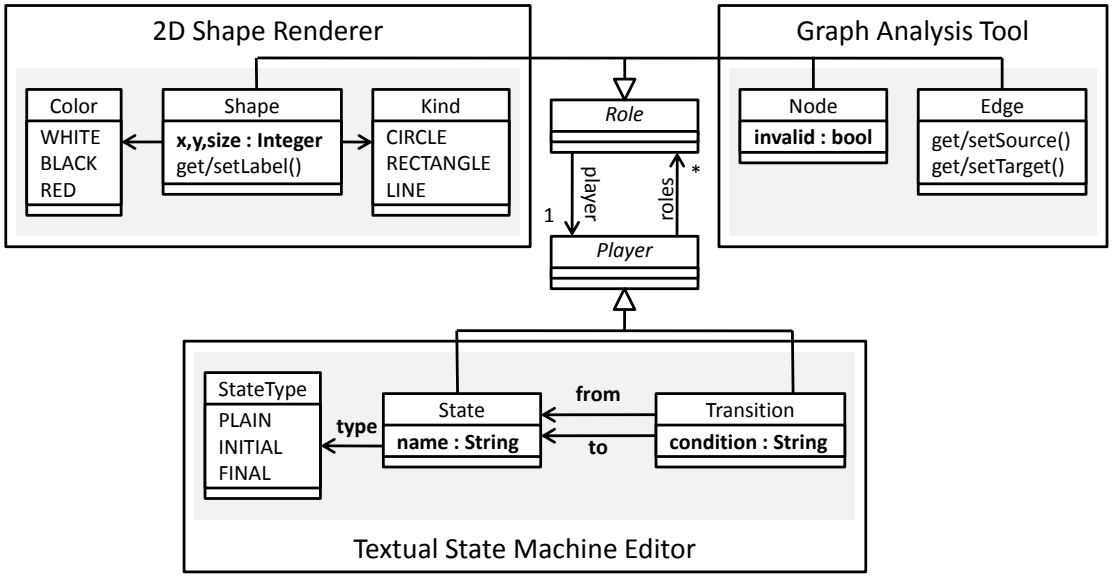


Figure 8.8: Simplified result of role composition.

To specify the physical representation of shared data we introduced the concept of `RoleGrounding` (cf. Fig. 8.6) to role composition. Grounding is used to identify role features that physically materialize data. Every feature that is tagged as being grounded by the role composition specification, must be realized as physical data store. This can be either a piece of memory or disk space.

The selection of grounded features, determines which data is physically shared across tools. Grounding features closes the delegation chains defined by the feature bindings. The latter specify how to obtain feature values from other roles. The former determines features that actually hold values and that are not derived. When tools access data, these requests will finally end up in reading or writing a grounded feature. However, tools do not know which grounded feature this will be. The physical representation of data is entirely decoupled from the interfaces used by tools to access data.

Given the role binding specification that was depicted in Fig. 8.7 and refined in Listing 8.1, one can derive a composed version of the three domain models. In Fig. 8.8 a simplified diagram showing the composed models can be found. The depicted class diagram is not sufficient to obtain an actual implementation of the composed role model. We will explain details on this in Sect. 8.4.3. But, to give an idea how the composition of role models is performed, this first simplified diagram should be sufficient.

Here, all role types are replaced by classes. Role types that were bound to one or more players (i.e., `Shape`, `Node` and `Edge`) do inherit from a new class `Role`, which represents the common superclass for all role types. Role types that are not played by other role

types (e.g., `State` and `Transition`) do inherit from class `Player`. In principle, all derived classes could inherit both from `Role` and `Player`, but to keep the example simplistic, we only added inheritance relations that are actually required.

The two additionally introduced classes—`Player` and `Role`—are connected by two references—`player` and `roles`. Through navigating these references, one can obtain all played roles and the player of each role. These two references are essential to implement the role binding, because they are used to obtain feature values by delegating requests to the role player. For example, if a client (e.g., the shape rendering tool) wants to retrieve the label of a given state, this request is forwarded to the player of the role type `Shape`—an object of type `State`. The same procedure applies to all other feature bindings. Requests that are issued to a role are forwarded to its player.

For grounded features, the situation is different. Instead of forwarding requests via delegation, these features are actually present in the respective classes. For example, the `x` and `y` features of class `Shape` are modeled as properties in the composed role model. Their values are stored in `Shape` objects.

One must always keep in mind that the composition of the role models is performed at tool integration time. This allows to postpone the decision on data materialization. In contrast, a priori integration requires new tools to adapt to existing and fixed data materialization. In our approach, tools are decoupled from the concrete materialization of data. Rather, the role composition model is adjusted to integrate new tools.

Figure 8.7 and Listing 8.1 depict the `RoleGroundings` and `FeatureGroudings` used to specify data materialization for our example. The example shows that grounding complements the flexibility of role binding as it allows for an integration-specific adjustment of data materialization. Data that shall be used by both the graph analysis tool and the shape renderer is materialized and shared using the roles in the state machine role model. Data which is only required within a specific tool (e.g., the `x`-, `y`, and `size` attributes of a `Shape`) is materialized in the respective role model.

Such flexible means to specify data sharing and materialization redeems tool developers from the challenge of anticipating extensibility required for future tool integration and avoid problems of data replication and synchronization.

8.4.3 Implementing Role Composition

Up to now, the languages to specify both the data abstractions used by tools (i.e., role models) and the possible interactions across them (i.e., role compositions) have been presented. The former specification—expressed in RoCoLa—did also include the decision about which data needs to be represented physically (i.e., the grounding).

To actually use role-based metamodeling for tool interoperability, the question how to implement both role bindings and groundings needs to be answered. As there is no single answer to this, because roles can be implemented in various ways (cf. [184] for an overview), we will present one feasible solution here. The aim of this is to show

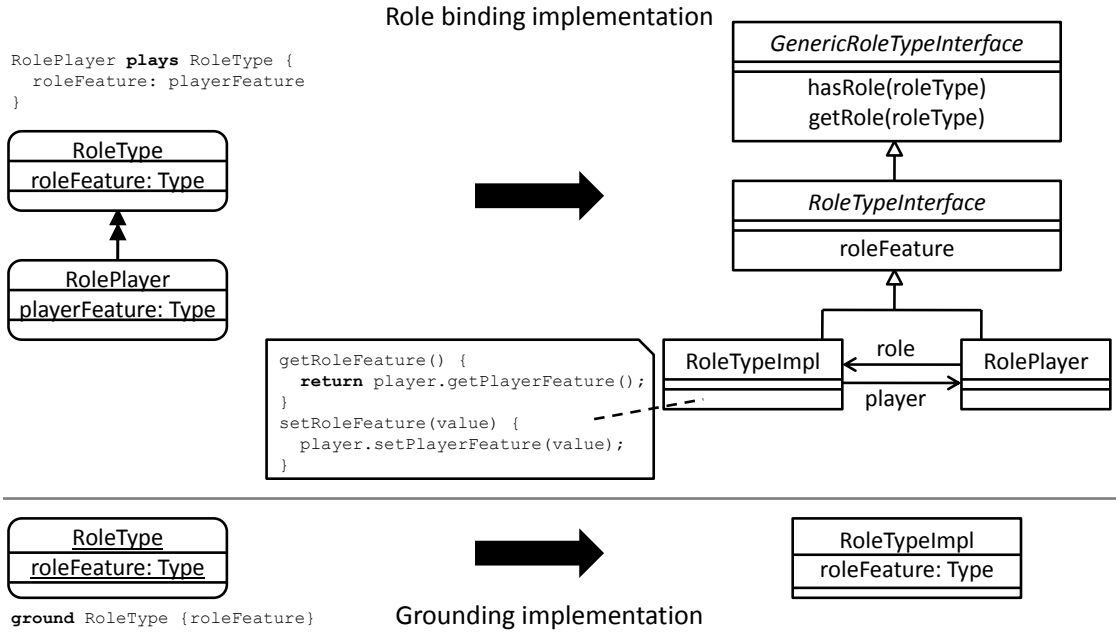


Figure 8.9: Object-oriented implementation of role binding and grounding.

that the presented approach can easily be implemented based on classic object-oriented technology. It does not raise the claim to be a universal solution.

Our mapping of roles, role bindings and groundings to object-oriented models is depicted in Fig. 8.9. The most simple mapping is the one for grounded roles and grounded features (i.e., attributes and references). These are basically mapped to plain classes and features respectively. As grounded roles and features are selected to be the part of the role model that is physically represented, this mapping is straightforward.

To explain the mapping of role types to classes, one must keep in mind, that role types can only exist in collaboration with one or more role players, where each player is connected to the role with a role binding. We map this relation to object-oriented models as shown in Fig. 8.9.

For each role type a dedicated interface (`RoleTypeInterface`) is introduced, which defines the features of the role. This interface is a supertype of both the implementation of the role type (`RoleTypeImpl`) and the player type. The latter inheritance relation reflects the fact that players need to provide all features that are expected by clients of the role. This pattern is quite similar to the Role Object Pattern [49]. We just added an additional interface `GenericRoleTypeInterface` to provide a meta object protocol [210] which can be used to check whether objects play a certain role (`hasRoleType()`) or obtain roles played by an object by role type (`getRoleByType()`).

The role binding specification presented before (cf. Listing 8.1), is used to fill the implementation of the `RoleTypeImpl` class. If the role binding is declarative (e.g., `label: name`), appropriate code needs to be generated to delegate calls to the methods `getLabel` and `setLabel` to `getName` and `setName` respectively. Bindings that use the imperative style (e.g., the binding for the color feature) must be translated to the target implementation language.

Depending on the style of the binding (i.e., either declarative or imperative), role bindings can be information preserving or not. Whether this is a requirement depends on the concrete domains that are bound to each other. Additionally, if the imperative style is used, one should ensure that `get` and `set` operations are inverse to each other.

A complete mapping of all role bindings, feature bindings and groundings results in a plain object-oriented model, which implements the integration defined by the role model and role binding specification. This object-oriented model can be derived fully automatically. If a different tool integration is required, the role binding and the grounding can be changed, leaving the involved tools untouched.

To obtain a sound integration of all tools, all role types and features must be either bound or grounded. This can easily be explained by the fact that the data required by tools must be either derived—by evaluating role bindings—or available in materialized form—in accordance to groundings.

8.5 Discussion

In this chapter, we presented a novel approach to establish support for RTE by explicitly anticipating future tool integration scenarios. We have illustrated the approach using a simple example synchronization scenario that involved three basic tools. To discuss the approach in detail, we want to clarify its limitations and differences. Most importantly, we want to give answers to the following questions:

- How does role-based tool integration differ from a posteriori approaches to tool integration?
- What is the impact of the expressiveness of the role binding language?
- How can existing data be migrated if the grounding specification evolves?
- Which technical requirements must be met to apply the approach?

The next subsections will try to answer these questions.

8.5.1 Comparison to A Posteriori Integration Techniques

Our role-based tool integration approach is explicitly designed to be applied a priori. We expect that different integration scenarios can occur in the future, and try to build

tools such that integration is rendered more easy. One important question that is raised by this course of action, is what can be gained in comparison to a posteriori techniques that do not plan for future integration, but merely first deal with this requirement at the time a new scenario is faced. Also, we need to determine the additional effort required to design for interoperability.

From our point of view there are multiple benefits that can be gained from explicitly expecting tool integration scenarios. First, a fair amount of redundancy can be completely avoided. Of course, this is the key to reduce synchronization effort, but it also comes with a nice side effect. Accessing data directly through role bindings, allows tools to process data concurrently. In contrast, if transformations are employed that create materialized views (i.e., target models), batch processing of data is required. After processing with one tool is complete, whole models need to be transformed before the next tool can be applied. If models grow large, this can become a bottleneck.

Second, our role-based approach to a priori integration allows for arbitrary tool interaction. Other mechanisms to support metamodel extensibility require the metamodel designer to foresee which interactions will be needed (cf. [204] for details). Thus, we do not only prepare the involved tools for a predefined set of integration scenarios, but for arbitrary combinations of tools and data models. We consider this a major improvement both over existing a priori and a posteriori techniques.

We have mentioned before, that basing tool integration on role models requires some changes in the tool design process. First, we must use role models instead of plain object-oriented metamodels to define the abstractions tools work on. Thus, for practical applications the absence of mature modeling tools with support for roles may pose a problem. Therefore, the additional effort for specifying tool domain models mainly depends on the availability of sophisticated tools.

Another activity where the effort between a priori and a posteriori integration can differ is the specification of the translation between domain models itself. Our approach requires role binding and grounding for that, while most a posteriori techniques require model transformations. The extent to which the effort for both approaches differs does therefore depend on the involved languages, for example, RoCoLa for our approach and QVTR in other cases. The effort is decreased if the used language provides appropriate concepts to specify the relation between domains. If both languages do so, the required effort should be comparable.

8.5.2 Expressiveness of Role Composition Languages

The role composition language that is used to obtain an integrated data model for all tools involved in an integration scenario, is an essential part of our approach. For example, the composition language RoCoLa provides concepts to map role features to player features, also on a conditional basis. Other composition languages may provide more complex feature operations. The expressiveness that is obtained from these features

has important implications w.r.t. the properties of the RTE systems that can be built. This can be explained as follows.

Role composition programs are similar to transformations in scenarios, where data is duplicated. Thus, role composition languages are somewhat equivalent to transformation languages. But, instead of transforming data to the required form, information is translated on demand. When a feature of a role type is requested by a client, the role composition is used to retrieve the respective information from the role player.

Compositions which are not information preserving pose the same implications as transformations which discard data. If data is passed to a role type, but neither passed to its role player, nor stored in a grounded feature of the role type, it cannot be accessed by other roles played by the same player. The data is discarded by the role composition. To avoid such situations, we recommend to use a role composition language, which does not allow to discard data. For example, if role features can be mapped to each other only (i.e., no operations are allowed to process them), this requirement is fulfilled.

The nice property about explicitly using a role composition language, instead of implementing ad-hoc bindings of tools to data repositories, is that one can analyze composition programs w.r.t. the preservation of information. By examining the used composition operators, one can detect situations, where information is discarded. Thus, one can make sure that the RTE scenario at hand does actually only discard irrelevant data. Also, one can always augment role types with additional grounded features to store data that would be discarded by the role composition.

In any case, the concrete features of the role composition language must be analyzed w.r.t. the preservation of data. For all language concepts that do not guarantee preservation, composition program designers must be warned about the implications.

8.5.3 Data Migration

Our role-based tool integration approach, relies on the separation of interfaces of tools from the physical representation of the data they process. By composing role models, one can integrate a set of tools in different ways. The composition controls which data is shared across tools and how this is done.

An important questions that is raised by this procedure, is how existing data can be migrated when the role composition model is changed, for example, when a new tool enters an integration scenario. In such a case, one must distinguish between changes that modify the way role features are grounded, changes that apply to existing feature bindings, and changes the merely establish new feature bindings or groundings.

The first case is problematic, because the physical representation of data may be altered. The second one may also raise issues, because feature values are obtained from other sources. Thus, the views held by the involved tools can change. Only, the third case—adding new feature bindings or groundings—is more easy to handle. This case corresponds to adapting a new tool to an existing data representation. The existing

bindings do not need to be changed here. However, to handle the first two cases—changing existing groundings and feature bindings—we see two options.

First, assuming that all views that are created by the definition of the role models, one could retrieve all stored data from the perspective of all tools individually. In our running example, the shape renderer, the graph analysis tool and the textual state machine editor would extract all data using their respective role model. The shape renderer would extract all shapes, the graph analysis tool all nodes and edges, and the state machine editor would retrieve all states and transitions. Then, the role composition can be replaced by a new one and all tools can put the extracted views into the system.

The second option is to analyze the changes between the old role composition model and the new one. For some cases, one can probably derive the required steps to adjust existing data to the new grounding. Which of the two options is more feasible requires future investigations.

8.5.4 Technical Requirements

Our role-based tool integration approach puts very few technical requirements forward. From our perspective, these cannot be considered as blocking factors when applying the approach, but we want to mention the requirements if only for completeness reasons.

First, the definition of domain abstractions (i.e., tool data structures) requires a role modeling language. We have presented such a language that is based on EMOF. If the approach is applied to other platforms, a similar language is required.

Second, code generation must be employed to obtain an implementation of the domain models. This can be either obtained by mapping to an object-oriented modeling language or directly to an object-oriented implementation based on a GPL (e.g., Java). We have presented patterns to realize this mapping in Sect. 8.4.3. Alternatively, one can directly employ a programming language with support for roles (e.g., ObjectTeams [211]).

Similar to the derivation of code that implements the domain model, code generators are needed to realize role bindings and role groundings. Since the approach as such is not bound to a specific technical platform, no particular requirements apply to the target language of the code generation process.

8.6 Summary

Using role modeling in the context of RTE was motivated by the need to integrate tools in order to share data and to interact in unforeseen ways. In particular the fact that such flexible data sharing cannot be achieved using classical object-oriented meta-modeling languages, drove this work. Existing a priori integration approaches did not allow tools to be independent of each other and interferes with tool-specific abstractions. A posteriori tool integration using transformations does not support data sharing and

impedes multilateral tool interaction. In the latter case, interoperability was not anticipated at all, while in the former it is anticipated only to a certain degree. This limited anticipation of tool interoperability at tool design time is one reason that renders RTE so difficult.

In this chapter we have shown how role-based metamodeling and role composition allow to handle arbitrary, unforeseen tool integration scenarios. We have used role models to capture the domain abstractions tools work on. Then, role composition programs are employed at integration time to bind domain abstractions to each other. A novel concept called grounding was used to specify which part of a role model is represented physically. All other role features need to be derived by examining role bindings.

While the idea of separating the data model required by tools from the physical representation of data is quite simple, explicitly providing modeling concepts to achieve this separation is not state-of-the-art. We have seen that roles and role composition are not only suitable to express such a separation, but furthermore allow to analyze the resulting tool integration. In particular the preservation of information is rendered possible by using a language dedicated to role composition.

The strength of our approach is that data models can be flexibly combined at tool integration time. One does no longer keep one (existing) data model fixed and adapt new ones to it. Both existing and newly added domain abstractions are treated equally.

One important issue that we have discussed, but not fully addressed yet, is the migration of existing data. Once a RTE scenario evolves (e.g., new tools are included), the grounding of data may change and existing data needs to be transferred to the new physical representation. To achieve such a migration an analysis of the changes made to the role bindings and grounding is needed.

The main limitation of our role-based metamodeling approach is that it needs to be applied during tool design time to prepare tools for interoperability. It cannot be applied as it is to integrate existing tools. Nonetheless, we strongly believe that the benefits gained at tool integration time outweigh enforcing tool developers to design their tools to allow for interaction. The fact that the proposed modeling approach allows to handle unanticipated future integration scenarios by construction relieves tool developers from anticipating unanticipated RTE scenarios.

9 Evaluation

This chapter contains the results of evaluating case studies for the different approaches to build RTE systems that have been presented in this thesis. Not all approaches were evaluated to the same degree. The most early approach underwent more investigations than the newer ones. Thus, the evaluation of the role-based and ontology-based synchronization approaches, being the approaches developed most recently, require further investigations (cf. Sect. 11.3). Nonetheless, we will try to highlight the positive and negative experiences that were gained while applying our approaches to practical synchronization problems in this chapter.

The backpropagation-based approach to RTE (cf. Chap. 6), being the first approach that was developed, is the most mature one and has been evaluated in different scenarios. In our initial publication [187], the approach was instantiated for the ISC technique and practically applied to synchronize both composed UML activity diagrams and Java programs. In later work that was performed within the research project MODELPLEX, the same ideas were evaluated on a composition system for UML and domain-specific security models. Details of this RTE system will be presented in Sect. 9.1. We selected this particular scenario, because it 1) has not been published before, 2) represents an industrial case study, and 3) outweighs the scenarios from [187] w.r.t. the complexity of the involved models.

Second, we evaluate a RTE system that employs backpropagation for a transformation that is not based on ISC. Namely, we apply our PREP approach to synchronize code and parameter models that are involved in template-based code generation. This is a particularly interesting scenario as it has high practical relevance. Using PREP to solve this problem will be presented in Sect. 9.2.

Third, we evaluate the ontology-based synchronization approach from Chap. 7 using a case study [212] that was previously performed using TGG rules (Sect. 9.3). We choose this scenario as it allows to validate whether the specification of synchronization rules in OWL and SWRL can handle mappings that are possible using the TGG formalism.

Our role-based tool integration approach (cf. Chap. 8) could not be applied to existing tools, because these do not employ role modeling. Thus, the evaluation of this approach was performed by building a new tool, which can benefit from the integration approach. More precisely, a refactoring tool was built and role models were employed to enable its applicability to arbitrary modeling languages. The initial results of this application have been published in [213] and will be recapitulated in Sect. 9.4.

All sections in this chapter follow the same outline. First, the context of the RTE scenario and an overview of the involved stakeholders and artifacts is given. Second, the application of our approach to build the RTE system is presented. Third, the pros and cons of the obtained system are discussed. The main focus of all evaluation sections is put on assessing more insights about the practical feasibility of our approaches rather than presenting all technical details that were required to accomplish the presented result.

9.1 Synchronizing Composed Security Models with Reuseware

In this section, an evaluation of our backpropagation-based approach to RTE (PREP) to a use case provided by an industrial research partner is presented and discussed. This work was jointly performed with Jendrik Johannes.

9.1.1 Scenario Description

During the MODELPLEX research project THALES, being one of the industrial partners, employed the Reuseware Composition Framework [188] to compose models. The use case provided by THALES involved different kinds of UML models to define a Systems of Systems (SOS), as well as a domain-specific language to specify security properties of system components. The goal of using Reuseware in this context was to obtain an integrated view of the SOS that would contain both the structure of the overall system, being defined in UML, and the security information contained in the security models. By combining both kinds of data, developers should get a more complete understanding of the system. The complete scenario is depicted in Fig. 9.1.

The involved models are represented by boxes and circles. Different kinds of borders indicate how artifacts are provided or derived. For example, the UML system models are created manually by users of the RTE system. Other artifacts (e.g., the integrated security model) are derived automatically. The background color of each artifact provides information about whether artifacts can be edited or not. Some of the artifacts—the Security DSL core model fragments and a model derivation specification—were manually developed to instantiate the Reuseware framework for the THALES use case.

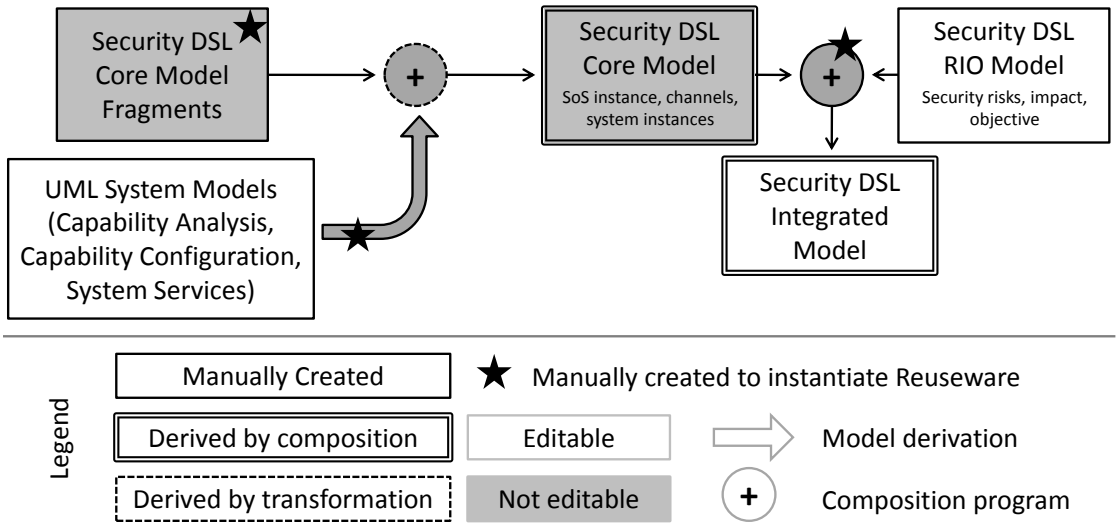


Figure 9.1: THALES round-trip scenario.

The models depicted in Fig. 9.1 are used by developers as follows. First, the structure of the SOS is captured in UML models. This structure includes the definition of individual systems (e.g., an air traffic control system) and their communication. Thus, the UML models do also contain parts that define which systems communicate with each other and which kinds of data they exchange.

Once such a UML model is available, the structure of the overall system is reflected in a *Security DSL core model*, which is automatically derived by Reuseware. This model resembles the same data as the UML model, but expresses it in a different language. To obtain this Security DSL core model, small model fragments are composed. For each pattern used in the UML model (e.g., a class to represent a system or a relation to represent a communication link) one such fragment was created manually. By assembling these fragments according to the contents of the UML model, Reuseware composes the Security DSL core model.

To annotate the derived Security DSL core model, a second security model—the *Risks, Impact, Objective (RIO) model* is used. This can be filled with security relevant data. For example, one can associate different kinds of risks with system components or define the impact a failure one of the systems may have. This data is substantially different from the structure of the overall system. One can consider it as an additional layer on top of the system structure.

To obtain an integrated view on the overall system, which was demanded by THALES, another composition program is required. This composition program needs to merge the structure of the system, being contained in the Security DSL core model, with the RIO model. To do so, the composition is designed to merge elements from both security

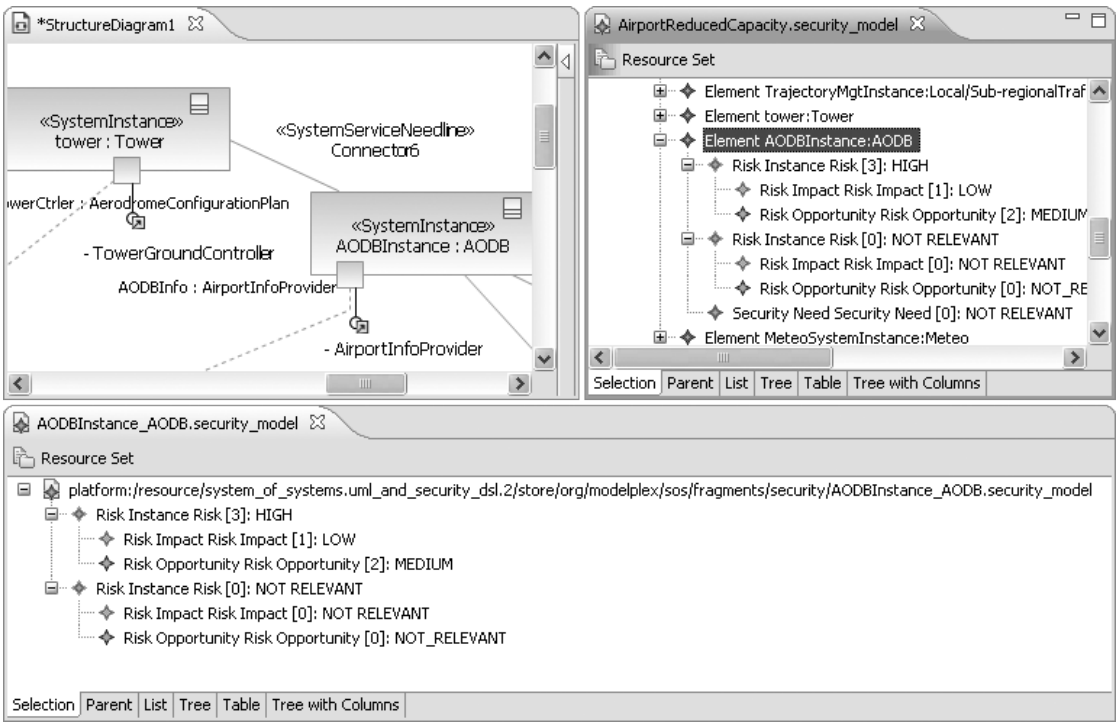


Figure 9.2: Example composition of models in THALES use case.

models by name. The resulting integrated model contains all information and can be used by developers to obtain a global picture of the system specification at hand.

An example composition of models from the THALES use case is shown in Fig. 9.2. One can find three models. The UML collaboration diagram (upper left part) and the RIO model (lower part) serve as input for the model composition. The model shown in the upper right part of Fig. 9.2 is the integrated security model. It contains the structural elements of the UML model (e.g., the `tower` and the `AODBInstance` element). In addition, data from the RIO model (i.e., the risk instances, impact and opportunities) can be found in the integrated model.

After Reuseware was successfully configured to perform all these composition tasks, it turned out that creating the overall view of the system was not yet sufficient. THALES quickly realized that changing the integrated view was an essential requirement for practical applicability. Thus, the composition of UML and security models turned into a full-fledged RTE scenario, where changes made to the integrated model were required to be handled gracefully. To meet this requirement, our backpropagation-based approach to RTE, presented in Chap. 6, was instantiated. The details of this endeavor are subject of the subsequent sections.

9.1.2 Applying the Approach

Prerequisites

For the THALES scenario most of the prerequisites to apply PREP were already met. All involved artifacts were represented as object-oriented models and respective EMOF-based metamodels for UML and the Security DSL were available. In addition, the artifacts required by Reuseware (e.g., composition programs) were also models as Reuseware itself was created according to the principles of MDSD [192]. Thus, no further activities were required to prepare the involved artifacts for the implementation of PREP.

Change Propagation

In Chap. 6, the running example used to introduce the PREP approach was ISC-based composition system for state machines. There, we observed that ISC makes heavy use of the `copy` operator when composing models. Elements from the input models are copied and reconnected to form the output of a model composition. We have also seen that propagating deletions and updates is rather easy if sufficiently detailed trace information is available. We had implemented support for traceability during our general investigations on RTE for ISC in [187]. Thus, deletions and updates of security-related model elements in the integrated system view were enabled by our earlier work. Whenever a security element was deleted or updated, the respective original element in the RIO model was changed accordingly.

The situation is different for deletions and updates that refer to the structure of the system. As shown in Fig. 9.1, one of the composition programs—the one used to create the Security DSL core model—is derived. To propagate changes back to UML models, the transformation realized by this composition program must be taken into account. But, since the composition program is obtained by a higher order transformation—a transformation that creates a transformation—propagating changes as sketched in Sect. 6.4.1 was not possible. Thus, THALES decided to exclude automatic handling of these modifications from the scenario. If the creation of the Security DSL core model would be implemented using a different transformation instead of employing Reuseware's metacomposition, one could extend the scenario to handle these kinds of changes.

Besides deleting and updating model elements in the integrated model, the scenario did also require support for adding new elements. For example, developers should be able to add new information about the impact of a system failure or associate new risks to system components in the integrated model. As discussed in Sect. 6.4.1 there can be multiple options how to propagate such additions. Multiple valid places where to add the new elements in the source models can exist.

In the THALES scenario, this was not the case. As it turned out during our investigations, there is a clear distinction between elements that belong to the Security DSL core model and the ones that belong to the ROI model. We were able to split the metaclasses

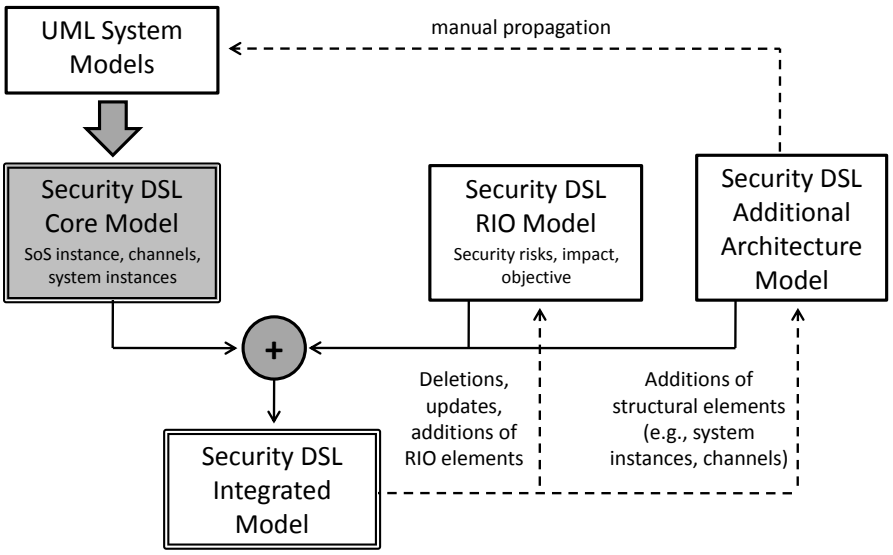


Figure 9.3: Change backpropagation in the THALES scenario.

of the Security DSL into these two disjoint sets. Thus, whenever an element is added to the integrated model, the decision where to put this new element can be made by examining the type (i.e., the metaclass) of this element. The scenario exposed type-based skeleton selection (cf. Sect. 4.1).

A summary of the change propagation strategy for the THALES use case is depicted in Fig. 9.3. Here one can find the models that were already introduced in Fig. 9.1, but also one new model—the *Security DSL additional architecture model*. The models are connected similarly as in Fig. 9.1. The Security DSL core model is derived from the UML system models and in a subsequent step composed with the ROI model. The result of this composition is the integrated security model. Change backpropagation paths are depicted by dashed arrows. As described above, deletions and updates on ROI elements (i.e., elements that are instances of metaclasses that belong to the ROI part of the security metamodel) are propagated to the ROI model. Also, additions of such elements are propagated in the same way.

The aim of the newly introduced model—the Security DSL additional architecture model—is to capture additions of model elements that encode system structure. For example, if a new communication channel is added to the integrated model, this is reflected by creating a channel in the additional architecture model. The reason for introducing this model was that propagating changes to the UML models was not possible. But, to enable additions of structural elements anyway, the new model was introduced. In practice, this requires to manually propagate additions from the architecture model (cf. Fig. 9.3) to the UML model, which THALES considered to be a feasible solution.

Replay

Replaying propagated changes in the THALES use case was quite easy to achieve. Since all transformations and compositions were fully automated, new versions of the integrated security model could be derived simply by executing the model composition on the changed input models. Since the use case did not require to evaluate multiple valid propagations, one does not even need to isolate the involved models. As there is always exactly one way to propagate a change, we simply perform this propagation and obtain a new integrated security model by executing the composition once again.

Synchronization Fitness Functions

PREP employs two important functions to control the selection of the most appropriate set of models after a change is applied by the user and propagated by the RTE system. The first function is called consistency fitness function and responsible to calculate the degree to which models are considered to be consistent. In the THALES case study even partially inconsistent models were not acceptable. Therefore, the consistency fitness function needed to be designed such that it returns 1 exactly if the composition of the input models yielded the expected, existing output model.

If we denote the UML system model as u, the ROI model as r, the additional architecture models as a, the integrated model as i, and the multi-staged composition that is realized by Reuseware as *compose*, a corresponding consistency fitness function can be defined as follows:

$$consistent(u, r, a, i) = \begin{cases} 1, & \text{if } compose(u, r, a) = i \\ 0, & \text{otherwise} \end{cases}$$

As in any other scenario where partial inconsistency is not allowed, this function is very simple. Moreover, the second function required by PREP—the conservation fitness function—is trivial. Because there is a distinct correlation between the type of newly added objects and the model they must be propagated to, multiple propagation options can never arise. There is always only exactly one way to propagate a change, which renders the conservation function obsolete. For completeness reasons we can define it to be a constant:

$$conserved(u, r, a, i, u', r', a', i') = 1$$

9.1.3 Discussion

Applicability

Using PREP in the THALES use case was quite successful. According to THALES, being able to make changes to the integrated model of the system was an important issue

that needed to be resolved to ensure the practical applicability of model composition. Since all involved artifacts were readily available as models conforming to EMOF, PREP could directly be applied. Moreover, once the relations between all involved models were clarified and it became clear how changes must be propagated, it took only a few days to implement the propagation strategy.

The support for RTE that was established in this scenario using PREP is not complete. Some modifications that users can apply to the integrated model cannot be handled automatically. While this seems to be a serious restriction from a conceptual point of view, it was not considered as a blocking factor by THALES. The partial support for editing and synchronizing models that was achieved, has been a huge step forward from a practical perspective. In addition, this restriction was caused by the metacomposition that was part of the use case rather than implied by PREP itself.

Gained Benefits

The model synchronization that was established in the THALES use case allows modelers to change the derived integrated system model within certain boundaries. Modelers can freely add and modify information that is purely related to aspects of security. Adding new structural elements to the modeled system is also possible, but requires manual adjustments of the UML models. This manual propagation is not mandatory, but required if the new structure shall be reflected in the UML models. Changing existing structural elements is not possible.

In summary, partial support to edit the integrated system model was gained by applying PREP. This is not completely satisfactory, but already a huge step forward, given that the THALES use case involved complex models provided by an industrial research partner. Thus, the feedback provided by THALES was very positive and the Reuseware Composition Framework including its support for RTE was elected the best tool in the MODELPLEX project.

Discovered Drawbacks

Beside the limitations w.r.t. the supported set of change operations, we also discovered other problematic points. First, change propagation is performed on the level of abstract syntax, but modelers use concrete syntax and respective editors to change models. Thus, we experienced the loss of layout in graphical editors after propagating changes. Also, some editors did not gracefully handle model changes. In particular, newly added elements were not automatically shown and in other cases editors forced a reload of the complete model after each change. This disturbs the editing process heavily.

EMF and the graphical editors that are built on top of its infrastructure do already aim at providing consistent editing processes. However, we observed that manual additions made to generated editors perished this idea in the THALES use case. Having editors

that can deal with model changes triggered from "outside" is essential to gain reasonable editing experience. One can say that PREP allowed us to synchronize models, but smoothly representing and embedding the effects of this synchronization in modeling environments is equally important.

Second, we propagated changes one at a time. Within a distributed environment this is not feasible anymore. We did not further experiment in this regard, but we expect serious problems if one wants to edit models concurrently. Also, executing the complete composition after propagating a single change interrupted the editing process. For the example models provided by THALES, the composition took only few seconds, but for even larger models the time to recompose models can become a bottleneck.

9.2 Synchronizing Generated Code and Parameter Models

In this section, we present and discuss an application of the backpropagation-based approach (PREP) to template-based code generation for object-oriented languages. This work was jointly performed with Marcel Böhme [214, 215].

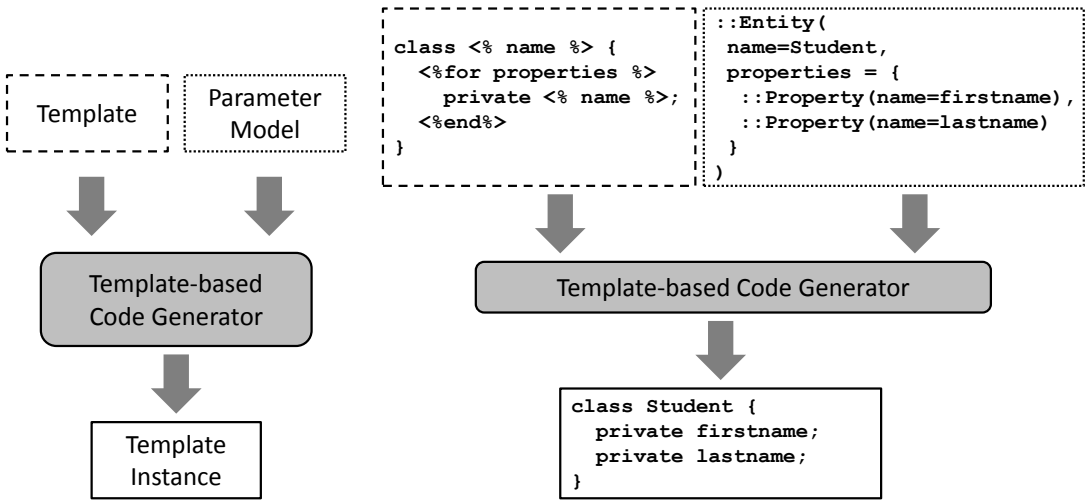
9.2.1 Scenario Description

In MDSD, abstract system descriptions are gradually refined to obtain a complete implementation. This refinement is accomplished by composing abstract descriptions with more concrete parts. It can be performed over multiple stages and using different kinds of artifacts. The MDA, being the OMG standard for applying MDSD, promotes the refinement of models until a complete system model is obtained. To eventually derive a runnable implementation from this model, code generation techniques must be applied.

Code generation techniques are transformations that use structured input (i.e., a model) and deliver code that is supposed to reflect the semantics of the input model. Usually, the obtained code adheres to some GPL such as Java or C. Often, code generation is referred to as metaprogramming, because code generators are programs that produce programs as output [216]. In the context of MDSD code generation is a particular type of model-to-text transformation [65].

One of the most prominent code generation techniques is template-based code generation. The basic idea of this approach is to provide stubs of the desired code and to fill these with values from parameter models. The stubs are called *templates*. In Fig. 9.4 this general idea is depicted. On the left side, in Fig. 9.4(a), the involved artifact types (i.e., templates, parameter models and template instances) can be found. The template instances are created by a template-based code generator.

Templates can contain arbitrary code of the target language (i.e., the language to be generated). In addition, templates can contain so-called metaelements. For example, templates may contain "holes", which are called placeholders (i.e., places where values are still missing). An example for such a metaelement can be found in the example



(a) Process (b) Example

Figure 9.4: General template instantiation process.

template depicted in Fig. 9.4(b). Here, the name of the class to be generated is obtained from the parameter model. The same applies to the names of the properties that are transformed to fields of the generated class.

In addition to placeholders, templates can contain conditional expressions and loops. The latter iterate over collections in the parameter model, while the former embed parts of the template depending on the value of some element in the parameter model. For example, the template in Fig. 9.4(b) iterates over all properties that are contained in the parameter model. One can say that template-based code generators compose their input (i.e., templates and parameter models) to form a template instance. Consequently, the latter is an amalgamation of the former. Every element that is part of a template instance either stems from the template or the parameter model. This does also imply that code generation introduces redundancies as data from the templates and the parameter models is replicated in template instances.

If code is generated from a perfect set of models and using a correct set of templates, there is no need to worry about the added amount of redundancy. Code is simply generated and the obtained system implementation can be used as it is. However, neither the models created in an MDSD process nor the involved code templates are perfect in practice. Rather, modelers make mistakes and metaprogrammers write incorrect templates. If the resulting failures are detected in the running system, either while testing or after deployment, code templates or models need to be fixed. This creates a serious problem for several reasons.

First, the running system uses the generated implementation (i.e., the template instances). Thus, faults are most easily detected in the code rather than in models or templates. Manually tracing the origin of a faulty piece of code back to the respective model element or template fragment requires a lot of effort. Second, if the origin of a failure has been isolated, the question is how to modify the model or the template in order to obtain a correct template instance. Removing faults would be substantially easier if one could simply change the template instance as required and derive the changes that are required for the parameter model or the template automatically. The goal of the resulting RTE scenario is therefore to keep templates, parameter models and template instances in a synchronous state.

9.2.2 Applying the Approach

Prerequisites

To apply backpropagation-based RTE as presented in Chap. 6 to the scenario at hand, a couple of prerequisites must be met. First, all involved artifacts must be represented as models. To do so, we both need metamodels for all involved languages (i.e., the target language, the template language and the parameter language) and means to instantiate concrete models. Here, the parameters are easy to deal with as they are already represented as models when generating code in the context of MDSD. Lifting the target and the template language to the modeling level requires more effort. In [217], we have shown how to perform such a lifting for Java, which was later used in [215] to create model-based template languages. A condensed version of the model-based template instantiation process from [215], can be found in Fig. 9.5.

Here the various metamodels for the involved languages are depicted, as well as the components that process concrete models. The conversion between abstract and concrete syntax is performed using generated bridges for the context-free grammars of the languages (cf. Sect. 5.3). The details on how to build these bridges for a concrete target language and the corresponding template language can also be found in [215, 217].

Once all involved artifacts can be represented as models, the process of instantiating templates can be lifted to the modeling level. Consequently, one can use a model-based code generator. Such a generator composes the input models (i.e., the template and the parameter model) to obtain an output model (i.e., the template instance). By operating on models the result of this composition is ensured to be syntactically correct. For details on how to implement such a model-based code generator, we kindly refer to [215], where a generic generator is described that can be used for arbitrary target languages.

Besides the syntactic correctness that is achieved here, one can trace the execution of the model instantiation process. Thus, for each model element in the output model its origin is known. Since model elements do either stem from a template or the parameter model, these are the only two possible origins. This situation is depicted in Fig. 9.6.

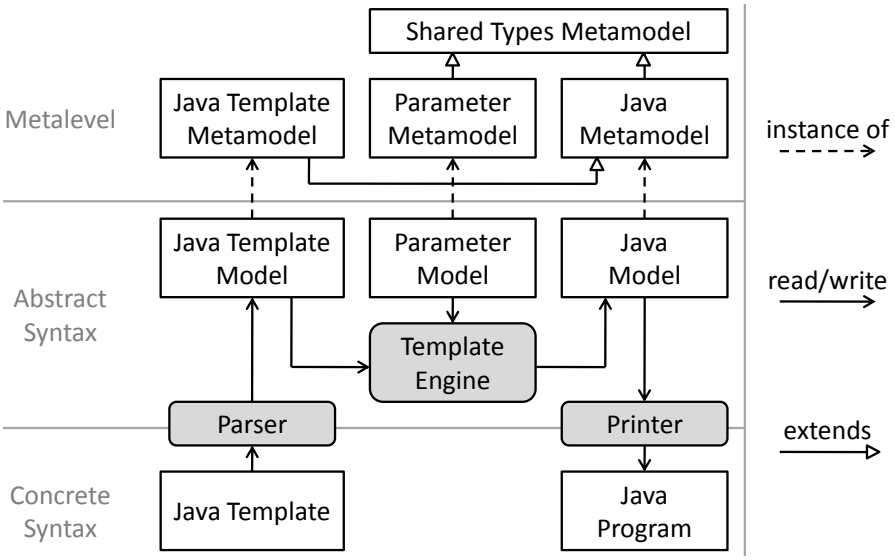


Figure 9.5: Model-based template instantiation process.

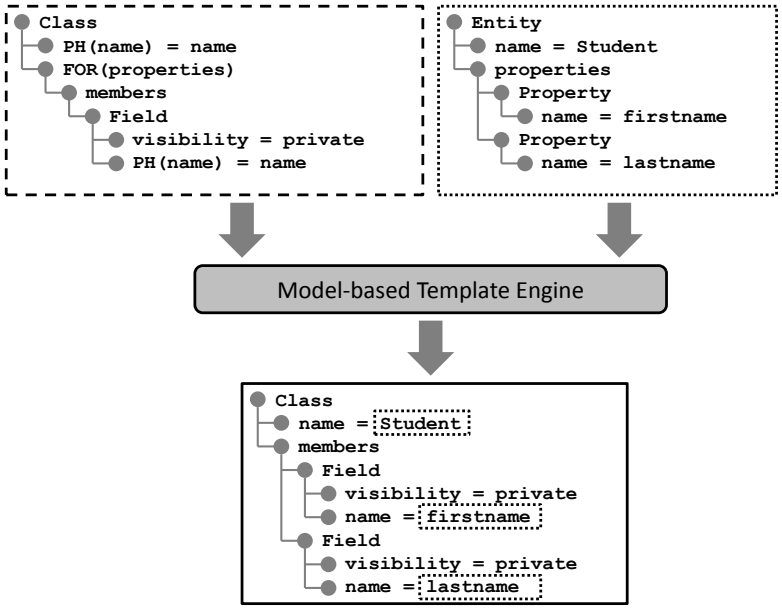


Figure 9.6: Model-based template instantiation example.

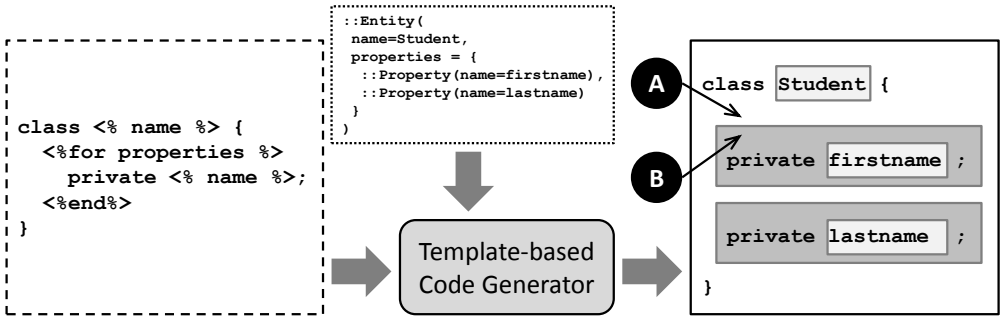


Figure 9.7: Multiple valid interpretations when inserting an element.

Here, the same template and parameter model as shown in Fig. 9.4 are instantiated, but this time the instantiation is performed purely model-based. Thus, Fig. 9.6 uses abstract syntax of the involved artifacts instead of concrete textual syntax as in Fig. 9.4. One can observe that the template model (upper left) is filled with values from the parameter model (upper right) to obtain the template instance (lower center). Most of the content of the template instance does in this case stem from the template. The only parts of the template instance model that are introduced by the parameter model are the name of the class (`Student`) and the names of the two fields (`firstname` and `lastname`).

Change Propagation

The model-based template instantiation process allows to obtain very detailed trace information. For each element in the template instance model, its origin is known. Thus, we can use these traces to propagate changes made to the template instance back to the original models (i.e., either to the template or to the parameter model). For example, if the name of a class is changed in the template instance, one can propagate this change to the parameter model as we know that the origin of this name was the parameter model. On the contrary, if the visibility of a field is changed in the template instance, we must propagate this change to the template, because the visibility value was defined there.

Similar to the running example in Chap. 6, the trace information is most useful for updates and deletions, because these two operations refer to existing elements. If new elements are added to the template instance, the situation is more subtle. Here, one must decide whether the new element must be added to the template or the parameter model. In addition, there can be multiple valid points where to insert a new element, which is depicted in Fig. 9.7.

Here, a case where a new element (e.g., a new field) is added to the template instance. More precisely, the new field shall be added before the first field that was created by the original template instantiation. To understand why there is multiple propagation

options, we have added gray boxes indicating regions that correspond to metaelements in the template. The light gray boxes correspond to placeholders, whereas the dark ones represent iterations of the for loop. We can track information about these areas due to the model-based instantiation technique.

If a new field is added, two interpretations (`A` and `B`) are equally valid. First, the field can be added before the loop (`A`). In this case, the addition must be propagated to the template by adding the new field before the loop. Second, the field can be assumed to be added inside the loop (`B`), which would result in an extension of the for loop body in the template. Since the regions (i.e., the gray boxes) are not visible when changes are applied to the template instance, one cannot be sure which of the two cases is intended by the user.

The template shown in Fig. 9.7 is quite simple. It contains only few metaelements. If templates contain more conditions or loops, the number of regions increases and thus can the amount of potential propagation options. In particular, if such elements appear deeply nested, the number of propagation options grows. Nonetheless, we can use the trace information to derive all valid propagation options and the decide—either manually of automatically—which one is most suitable.

Replay

Implementing the replay step for the scenario of template-based code generation is quite easy. To replay the transformation we basically need to instantiate the template once again. We simply provide the changed input models (i.e., the template and the parameter model) to the model-based template engine and obtain a template instance model that reflects the propagated changes.

Similar to the examples presented in Chap. 6 and Sect. 9.1, one must ensure that the replay step is performed in isolation. Consequently, a separate set of input and output models must be maintained to ensure that replaying template instantiation does not interfere with the original models.

Synchronization Fitness Functions

In Sect. 6.4.3 we introduced the notion of synchronization fitness functions. These functions are employed to evaluate which propagation option is most suitable, given that multiple such options exist. Indeed, this situation can occur for template-based code generation, because adding new elements can result in multiple propagation choices. Thus, the question is how the synchronization fitness function can be designed to pick the most feasible option.

The first type of function that is required here is the consistency fitness function. The aim of this function is to determine the degree to which the artifacts involved in a RTE scenario are consistent. Since we cannot accept partially inconsistent artifacts in the

case of template-based code generation, our consistency fitness function must reflect this requirement. If we denote the template as t, the parameter model as p and the template instance as ti, we can define this functions as follows:

$$consistent(t, p, ti) = \begin{cases} 1, & \text{if } instantiate(t, p) = ti \\ 0, & \text{otherwise} \end{cases}$$

Here, $instantiate(t, p)$ refers to the function that is implemented by the model-based template engine. This consistency fitness function states that models are only considered to be consistent if the output of the instantiation process is exactly the template instance. This definition corresponds to the one used in Sect. 6.4.3 and is straightforward.

The second type of function—the conservation fitness function—is more important here as it controls which propagation option is selected. When designing this function one will observe that no single function can be used in all code generation scenarios. To illustrate this, consider a very basic state-based conservation function, which computes the relative similarity between the original models and the changed ones (cf. Sect. 6.4.3 for the original definition of $rsim$):

$$conserved(t, p, ti, t', p', ti') = \frac{rsim(t, t') + rsim(p, p') + rsim(ti, ti')}{3}$$

This function computes the similarity of all involved models and derives the average similarity by adding up all individual similarities and dividing them by the total number of similarity values. The function differs from Def. 10, because the latter assumed that there is only one input model, whereas our use case has two input models—the template and the parameter model.

If one uses this conservation fitness function, the propagation option that implies the fewest number of overall changes is favored. For example, consider the insertion options that were shown in Fig. 9.7. In case `A` the new field was added before the loop region. Thus, it will be added before the loop (in the template) rather than inside the loop. In case `B` the new field was assumed to belong inside the loop region. Therefore, the latter option would trigger one change per loop iteration, while the former implies only a single change. Our basic conservation function would therefore prefer interpretation `A`. However, this might not be the intended behavior. Sometimes it can be more reasonable to accept a propagation option that implies many changes in the template instance.

A similar situation can be observed when an element which corresponds to the body of a conditional expression in the template is deleted. Here one can either delete the body of the expression from the template or change the value in the parameter model that enabled the embedding of the respective element in the template instance. Both options yield the same result (i.e., the same template instance), but if we consider the involved template to be quite mature, changing the parameter model can be more appropriate.

The two examples illustrate the main problem of the given conservation fitness function. Depending on the concrete code generation scenario one must favor changing one

artifact over another. For example, if a set of mature templates is used, changes to the parameter model should be favored. In contrast, if the parameter model is considered to be stable, changes in the template may be more appropriate. Introducing weights to the conservation fitness function can support this demand. Instead of treating all changes equally, we can compute a weighted sum that favors changes to one kind of artifact over another one. The resulting function can be defined as follows:

$$conserved(\ldots) = \frac{w_t * rsim(t, t') + w_p * rsim(p, p') + w_{ti} * rsim(ti, ti')}{w_t + w_p + w_{ti}}$$

This function is a generalization of the first variant where all weights (w_t, w_p and w_{ti}) were 1. By choosing concrete values for the weights, preferences for changing a particular type of artifact can be realized. One cannot give a general "best setting" here, because the most valuable set of parameters depends on the maturity of the involved artifacts and the preferences of the developer.

9.2.3 Discussion

Applicability

In this section, we evaluated PREP in the context of template-based code generation. We represented all involved artifacts (i.e., templates, parameters and target language code) as models. Since template languages are usually extensions of a GPL, being a complex language on its own, this is not trivial. Since we could reuse a model-based implementation of the Java language from earlier work [217, 218], this step was easy to realize. However, if no such model-based representation is available, this is a obstacle.

Once all involved artifacts were lifted to the modeling level, the actual application of PREP could begin. To obtain fine-grained trace information, the model-based template engine needed to be extended. This requires some technical effort, but does not pose a big problem. Instantiating templates requires the `copy` operator, which was extended to store trace information. Also, template metalanguages provide few concepts (i.e., placeholders, loops and conditional expressions). Thus, extending the interpretation of these concepts with support for traceability is also easy to accomplish.

Implementing the actual propagation of changes was more complicated. Templates use textual syntax, which implies that edits are performed different from editing graphical models. In graphical editors users explicitly apply individual changes. This yields a series of models that are connected by a sequence of edit operations. On the contrary, textual editors provide a series of models. The edit operations are not explicitly known. Rather, they must be computed before changes can be propagated. This is possible, but less accurate in comparison to model editors where users perform changes explicitly.

In summary one can say that applying PREP to template-based code generation involves significant effort. In particular the fact that textual artifacts are often considered

to be conceptually different from other models, renders this task difficult. We have performed research trying to invalidate this point [178, 217], but there are still open questions. For example, it is not clear to what extent the computation of model differences hinders RTE. We do know that explicit change information is more accurate, but we cannot give definite statements about the implications of inaccurate change data.

Gained Benefits

Applying backpropagation-based RTE to template-based code generation showed that simple changes can be propagated from template instances to templates and parameter models. For example, changing names that are copied from a parameter model can be supported. Also, changing static content that is copied from templates is possible. While this does certainly not cover all potential edit operations, it is a significant improvement compared to classic template instantiation which is purely forward-oriented.

Besides the support for a restricted set of edit operations, we obtained fine-grained trace information. This information is created by the model-based template engine and cannot be used for backpropagation only. If system failures are assigned to a certain piece of code, one can find the elements in the parameter model or the template that account for the creation of this code fragment. Thus, developers are equipped with a tool that assists in fixing system faults, even if no automatic propagation can be provided.

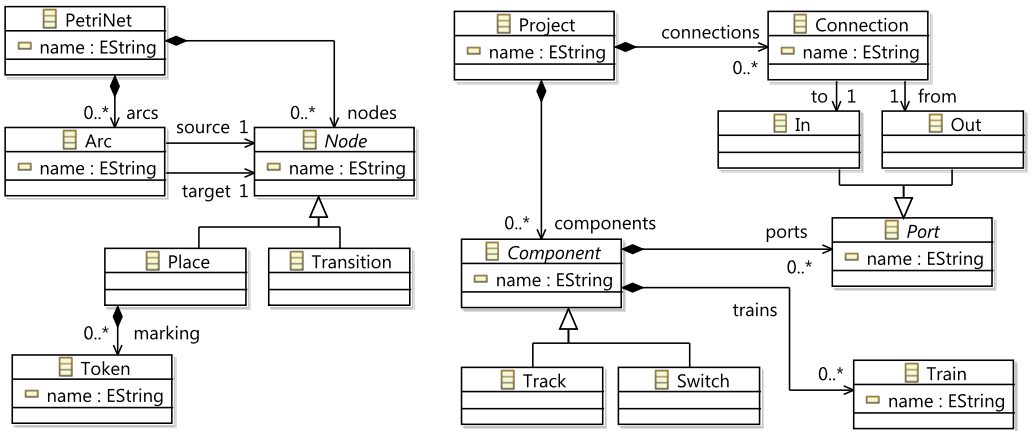
Discovered Drawbacks

Besides the effort to establish RTE support for template-based code generation, we discovered more problems that require special attention in future research. First, our template languages are restricted w.r.t. their expressiveness. This is in contrast with existing template engines that are often Turing complete. Most prominently we do not offer operators to perform computations on primitive types (e.g., to concatenate strings). If one requires such operators, a specification for the backpropagation of changes is required. For many operations on primitive types this is not possible.

One way to work around this issue, is to provide RTE support for all parts of the generated code where backpropagation is possible. Changes applied to other parts (e.g., names that are obtained by string concatenation) need to be handled manually. While this is not a fully automated solution, it will at least provide partial RTE support.

Alternatively, one can separate code generation into two phases. First, one must compute all derived values (e.g., new names) from the parameter model and create a new, intermediate parameter model. Second, the contents of the intermediate model are composed with the template to form the template instance. This separation is proposed in [133] and allows to handle the two aspects of code generation individually.

Second, the number of valid choices to propagate a single change can easily grow large. In particular, if metaelements (e.g., loops and conditional expressions) are deeply nested,



(a) Metamodel for Petri nets

(b) Metamodel for toy train models

Figure 9.8: Metamodels for domain-specific models in evaluation scenario.

their boundaries overlap. Thus, if code is added next to a set of overlapping boundaries, multiple propagation choices do exist. Often, picking one of them cannot be automated as all choices are equally valid. In this case developers need to explicitly pick a choice. This is a drawback in the sense that full automation cannot be achieved here. It is not a downside of PREP, but rather a property of this particular scenario.

9.3 Synchronizing Domain-specific Models using OWL and SWRL

In this section, we evaluate the ontology-based approach to RTE that was presented in Chap. 7 using an example scenario from [212]. This work was jointly performed with Federico Rieckhof and parts thereof have been published in [199].

9.3.1 Scenario Description

To evaluate the ontology-based RTE approach, we used a scenario where two domain models need to be kept in sync. This scenario is taken from [212]. The involved domains are *Petri nets* [219] and *toy trains*. The goal of the synchronization is to make sure the Petri net model implements the dynamic semantics of the train model.

The metamodel for Petri nets is shown in Fig. 9.8(a). Each Petri net consists of a set of arcs and nodes. Arcs connect nodes in a given direction. Nodes are either places or transitions. The former can hold multiple tokens. A constraint, which is not depicted in Fig. 9.8(a), forbids that arcs connect two places or two transitions. The types of the

Name	Petri net elements	Toy train elements
Petrinet2Project	`PetriNet`	`Project`
Place2Out	`Place`	`Out`
Transition2In	`Transition`	`In`
Connection2Arc	`Connection` (from `Out` to `In` port)	`Arc` (from `Place` to `Transition`)
Track2Arc	`Track` (from `In` to `Out` port)	`Arc` (from `Transition` to `Place`)
SwitchA2Arc	`Switch` (with one `In` and two `Out` ports)	Two `Arc`s, one `Transition`, and two `Place`s
SwitchB2Arc	`Switch` (with two `In` and one `Out` port)	Three `Arc`s, one `Transition`, and one `Place`

Table 9.1: Mappings between Petri nets and toy train models.

connected elements must be different. In addition, Petri nets, arcs, nodes and tokens have a name.

The metamodel for toy trains is depicted in Fig. 9.8(b). Each toy train project consists of components and connections. Components can be either tracks or switches and expose ports. Connections connect ports. Here, the direction is given by making a distinction between ports of type `In` and `Out`. Tracks have exactly one incoming and one outgoing port, whereas switches can have two ports of the same type (i.e., either two `In` and one `Out` port or vice versa). Furthermore, trains can reside on tracks and switches.

The mapping between the two domains can be put as follows. Each Petri net relates to one toy train project. Each track relates to an arc. `Out` ports relate to places and `In` ports to transitions. Similar to tracks, connections between ports are related to arcs. A more complicated mapping is needed for switches. Switches with two incoming ports relate to an arc connecting a transition and a place. Switches with two outgoing ports relate to two arcs that connect a single transition with two places. In total, six mappings are required, which are summarized in Table 9.1.

To give an example of a pair of synchronized models, consider the models that are shown in Fig. 9.9. The figure uses concrete graphical syntax for both models. To make the mapping more obvious the components of the toy train model (shown in Fig. 9.8(b)) are also depicted in gray shade in Fig. 9.9. This way it is easier to see which parts of the Petri net correspond to which parts of the train model.

As the example is taken from a paper on the TGG formalism [212], one can derive that the synchronization at hand can be solved using a set of bidirectional transformation rules. Now, the question is, to what extent ontology tools can perform the same task.

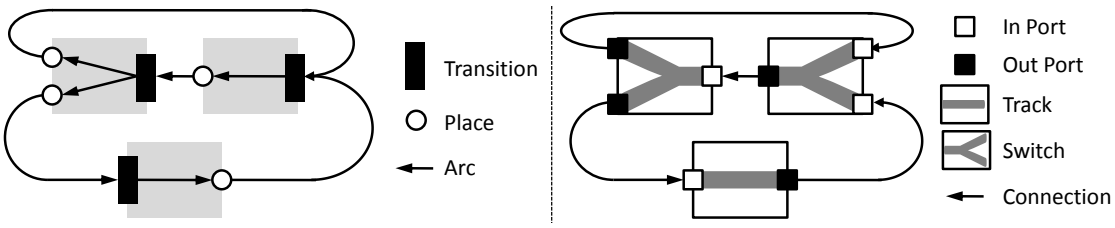


Figure 9.9: Example of synchronized Petri net (left) and toy train model (right).

9.3.2 Applying the Approach

Prerequisites

To use tools that are capable of reading and processing OWL ontologies and SWRL rules, the metamodels from Fig. 9.8 must be translated to respective ontologies. To do so, we used the OWLizer tool developed within the MOST project[1]. The OWLizer takes Ecore metamodels and produces corresponding ontologies in OWL Manchester syntax. As discussed in Sect. 5.4, we obtain disjoint OWL classes for all metaclasses, object properties for all defined references and data properties for all attributes. In addition, the OWLizer is capable of transforming models to sets of OWL individuals and corresponding facts. Thus, transferring metamodels and their instances to the ontology space was easy to achieve.

The complete ontologies of both metamodels, as they are produced by the OWLizer, can be found in Appendix A2. For the sake of readability, we will omit the prefixes for all features that have unique names in the following. For example, we will use `source` instead of `Arc_source`, as there is no second feature named `source`.

Synchronization Rules

To capture the synchronization rules between Petri nets and toy trains models we used both OWL subclass definitions and SWRL rules. The former were employed to map Petri net elements to toy train projects, places to `Out` ports, and transitions to `In` ports. Thus, they cover the mappings *Petrinet2Project*, *Place2Out* and *Transition2In*. The concrete subclass definitions can be found in Listing 9.1.

```
1 Class: Petrinet2Project  EquivalentTo: PetriNet, Project  SubClassOf: owl:Thing
2 Class: Place2Out         EquivalentTo: Place, Out         SubClassOf: owl:Thing
3 Class: Transition2In     EquivalentTo: Transition, In     SubClassOf: owl:Thing
```

Listing 9.1: OWL subclass definitions for Petri net and toy train models.

[1] http://www.most-project.eu

The definition of these subclasses is straightforward, because there is a simple 1-to-1 mapping between these classes. To express the more complex relations in the evaluation scenario, we employed SWRL rules. In particular, if multiple model elements must form a pattern to represent one or more concepts of the opposite domain, SWRL rules are feasible. The mappings *Connection2Arc*, *Track2Arc*, as well as *SwitchA2Arc* and *SwitchB2Arc* are examples for such a cases in our evaluation scenario.

To illustrate how rules for these mappings can be designed, let us first consider the *Track2Arc* mapping. Here, the fact that whenever a Petri net contains an arc that connects a transition to a place, a corresponding track must exist, can be expressed using the following rule:

$$\begin{aligned} &Arc(?x) \wedge Place(?y) \wedge Transition(?z) \wedge \\ &target(?x, ?y) \wedge source(?x, ?z) \rightarrow \\ &Track(?x) \wedge Out(?y) \wedge In(?z) \wedge \\ &ports(?x, ?y) \wedge ports(?x, ?z) \end{aligned} \tag{9.1}$$

One can read rule 9.1 as follows. All individuals of type `Arc`, which are related with one place and one transition by the object properties `target` and `source`, imply that the individuals of type `Arc` are also of type `Track`, the individuals of type `Transition` are also of type `In` and that the individuals of type `Place` are also of type `Out`. Furthermore, the object property `ports` must connect the track with its ports.

As rule 9.1 does only work in one direction, a second rule is required to express that each track and its respective ports must be mapped to an arc connecting a transition and a place in the Petri net. The rule to express this is as follows:

$$\begin{aligned} &Track(?x) \wedge Out(?y) \wedge In(?z) \wedge \\ &ports(?x, ?y) \wedge ports(?x, ?z) \rightarrow \\ &Arc(?x) \wedge Place(?y) \wedge Transition(?z) \wedge \\ &target(?x, ?y) \wedge source(?x, ?z) \end{aligned} \tag{9.2}$$

As one can see, both a forward and a backward rule is needed to synchronize individuals of type `Track` from the toy train metamodel. Using rule 9.1 and rule 9.2, a reasoner is able to infer the synchronization in both directions. The need for two rules confirms that SWRL rules can only work in one direction, that is, they are not equivalent to bidirectional model transformations. Rather, they provide means to realize unidirectional incremental transformations.

The rules 9.1 and 9.2 establish the *Track2Arc* mapping. If we closely inspect the mappings between the two domains (cf. Table 9.1), we can find other mappings (e.g., *SwitchB2Arc*) that can interfere. Similar to tracks, switches with two incoming ports

are mapped to an arc that connects a transition to a place. But, not only switches are mapped to arcs. Connections between toy train components are also mapped to arcs (*Connections2Arc*). Obviously, we need to distinguish between arcs that represent tracks, arcs that correspond to switches and arcs that are related to connections.

To do so, we must take the relations of the arcs into account. The *Connection2Arc* mapping applies exactly if arcs connect places to transitions. If arcs point in the opposite direction (i.e., from transitions to places), they must be mapped to a track. However, one must be careful here, because this applies only if the respective transition does have at most one outgoing and at most one incoming arc. Otherwise, the transition is part of a pattern that corresponds to a switch.

To ease the specification of SWRL rules that implement this logic, we decided to add two helper rules, which compute whether a transition has multiple incoming or outgoing arcs. The result is then stored in a data property `hasMultipleIncomingArcs` and `hasMultipleOutgoingArcs` respectively. The two helper rules are as follows:

$$\begin{gathered} Transition(?t) \wedge Arc(?a1) \wedge Arc(?a2) \wedge \\ target(?a1, ?t) \wedge target(?a2, ?t) \wedge \\ differentFrom(?a1, ?a2) \rightarrow \\ hasMultipleIncomingArcs(?t, true) \end{gathered} \tag{9.3}$$

$$\begin{gathered} Transition(?t) \wedge Arc(?a1) \wedge Arc(?a2) \wedge \\ source(?a1, ?t) \wedge source(?a2, ?t) \wedge \\ differentFrom(?a1, ?a2) \rightarrow \\ hasMultipleOutgoingArcs(?t, true) \end{gathered} \tag{9.4}$$

Using these two helper rules, one can specify all remaining mappings without any conflicts. To give an example, consider the rule for mapping arcs to switches of type A (i.e., switches that have two `Out` ports):

$$\begin{gathered} Arc(?x1) \wedge Arc(?x2) \wedge Transition(?z) \wedge \\ Place(?y1) \wedge Place(?y2) \wedge \\ source(?x1, ?z) \wedge source(?x2, ?z) \wedge \\ target(?x1, ?y1) \wedge target(?x2, ?y2) \wedge \\ hasMultipleOutgoingArcs(?z, true) \rightarrow \\ Switch(?x1) \wedge Out(?y1) \wedge Out(?y2) \wedge In(?z) \wedge \\ ports(?x1, ?y1) \wedge ports(?x1, ?y2) \wedge ports(?x1, ?z) \end{gathered} \tag{9.5}$$

Here we use the data property `hasMultipleOutgoingArcs` to make sure that only patterns are matched that actually correspond to switches of type A.

9.3.3 Discussion

Applicability

Applying our ontology-based model synchronization approach to synchronize Petri nets and toy train models was feasible. We were able to realize all required mappings using OWL subclass definitions and SWRL rules. However, the mappings that were formalized using the latter specification mechanism, required two rules—one for each transformation direction. For the scenario at hand, we did not experience any problems w.r.t. expressivness, but this may not be the case for other synchronization applications.

Transferring models to the ontological space was easy since the existing OWLizer tool provided the required functionality. Both Ecore metamodels and models were translated to respective OWL ontologies.

Gained Benefits

During the design of the SWRL rules, we discovered one major benefit that can be gained from employing DL. We successively extended the set of rules and the example models that were used to check the correctness of the synchronization. Throughout this process, the Pellet reasoner detected inconsistencies among the set of rules—multiple times. For example, first sets of our rules were conflicting, because the constraints of the number of incoming and outgoing arcs for transitions were missing. Both rules to map arcs to tracks and to map them to switches were applicable. Thus, the reasoner inferred both types and issued an exception, because the types `Track` and `Switch` are declared to be disjoint. We found this consistency checks for rules particularly useful.

For other model transformation languages, the very same problem can be observed if rules overlap. If multiple transformation rules expose patterns which are not disjoint, multiple rules can match the same input. This is usually resolved by assigning priorities to rules, but often not detected automatically. More sophisticated transformation formalisms allow to perform a critical pair analysis [220] to detect such problems instantly (i.e., before the actual execution of the rules).

Discovered Drawbacks

Having said that the detection of inconsistencies in our ontologies was beneficial, we must add that the explanations given by the reasoner were not always very helpful. Pellet gives the set of all axioms that cause the inconsistency, but in general this set can be very large. Thus, resolving the problem did often involve tedious inspections of the rules.

Moreover, ontologies were reported to be inconsistent if structural errors were present (e.g., if an object property is assigned to individual of the wrong type—a type that is not in the domain of the property). Such errors cannot occur when creating models, because their abstract syntax does not allow this. Ontology tools should similarly report such errors as syntactic problems rather than as inconsistencies.

9.4 Integrating a Refactoring Tool with Arbitrary Metamodels

In this section we elaborate on building a generic refactoring tool that can be used with arbitrary metamodels to show a practical application of our role-based tool integration approach. This work presented was performed jointly with Jan Reimann [213].

9.4.1 Scenario Description

Every software system is subject to continuous change. During the development earlier design decisions need to be reconsidered. When a system is deployed, its environment and thus the requirements that must be met change. Being able to safely restructure a system and thereby improve its design is therefore essential to preserve quality while a system evolves. This is the essence of Lehman's Laws of Software Evolution [221].

William F. Opdyke introduced the term *refactoring* for structural changes that improve the design of a software system, while keeping its semantics constant [222]. Since then, refactorings are recognized as an essential tool for developers [223] and all modern development environment provide refactorings for GPL programs.

With the advent of MDSD a new challenge w.r.t. improving the design of software systems emerged. Since MDSD processes employ different modeling languages (e.g., custom DSLs), refactorings are required for a variety of languages. Building refactoring tools that are specific to exactly one language only is still an option, but to reduce the overall effort for tool building, a generic refactoring tool would be more appropriate. Of course, such a generic tool must provide means to apply refactorings to arbitrary languages, but still account for the specifics exposed by individual languages.

Refactory is such a generic refactoring tool for EMF-based modeling languages [213]. It was built based on the idea of role-based tool integration and can therefore serve as an evaluation scenario. In this section, we discuss how role models and role bindings can be used to integrate a generic tool with arbitrary modeling languages.

9.4.2 Applying the Approach

Prerequisites

The role-based RTE approach requires all involved tools to operate on role models instead of plain object-oriented models. Thus, the main components of Refactory (e.g., the

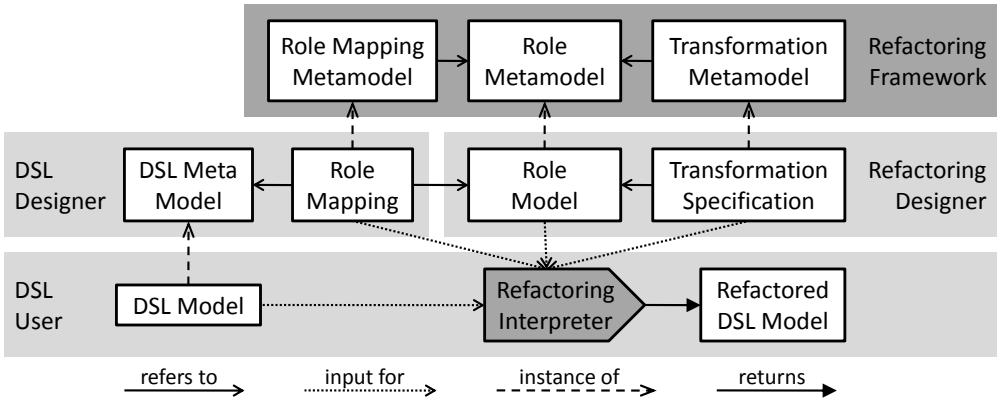


Figure 9.10: Refactory overview.

interpreter for transformation specifications) were designed to solely rely on role models. As Refactory was built from the scratch, this prerequisite could easily be met.

Role Modeling

In Chap. 8 we have argued that tool interoperability can be fostered by using role models to decouple the actual implementation of a tool from the physical representation of the data it operates on. The role model can be considered as an interface that is used by a tool and that must be provided by another tool or a data repository.

However, in the case of Refactory a single role model was not sufficient. To understand this, we must reconsider the aim of a refactoring tool. Its main task is to restructure a given system model. To perform this restructuring a description of the required individual transformation steps is needed. For example, a refactoring that extracts parts of a model to a new component must first create a new empty component and then move the selected parts respectively. We call this list of steps a *transformation specification*.

While designing such specifications one can quickly realize that the data model to operate on differs from refactoring to refactoring. For example, the extraction refactoring requires a container type—to create the new container—and a reference held by this type—to store the moved elements. Other refactorings (e.g., one that renames elements) do require a different data model. Thus, transformation specifications and role models form a pair. This relation is depicted in Fig. 9.10.

Here one can also see that the relation between concrete modeling languages—in Fig. 9.10 denoted as DSLs—is established by a *role mapping*. This mapping corresponds to the role bindings that were introduced in Chap. 8. The differing name is used for historical reasons. In Fig. 9.10 one can also find the different stakeholders who are involved in the definition of a refactoring. The *refactoring designer* creates role models

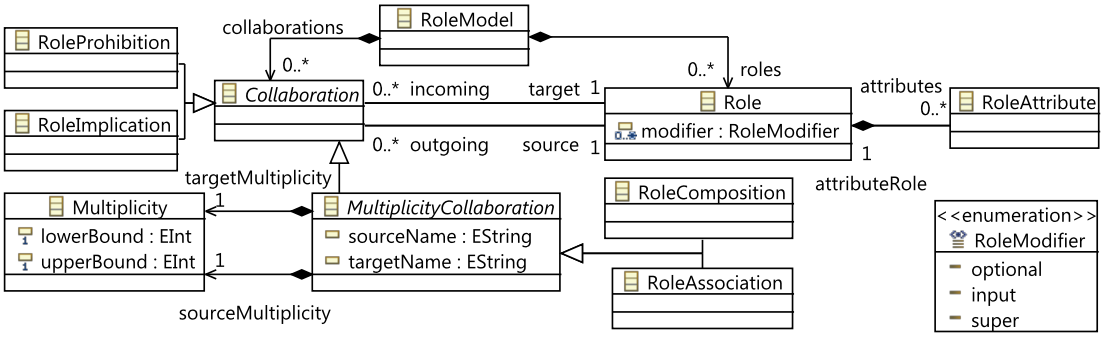


Figure 9.11: Refactory role modeling language metamodel.

and matching transformation specifications. The former are then subject to a mapping which is created by the *DSL designer* to bind a metamodel to a role model. This binding enables users of the DSL to actually apply refactorings.

Returning to the aspect of tool integration, we can state that a specific role model is required for each generic refactoring. One can consider the corresponding transformation specifications as individual tools, even though they are executed by the same interpreter. However, this is to reduce implementation effort only. Equivalently, each transformation specification could be realized by an individual tool, which would emphasize the idea that the number of tools corresponds to the number role models a little more.

To define role models in Refactory, a custom role modeling language was designed. This language is shown in Fig. 9.11 and provides common role modeling concepts. For example, role types (metaclass `Role`) and role features (metaclasses `Collaboration` and `RoleAttribute`) can be found in this metamodel.

In addition to the general concepts that all role modeling languages have in common, the metamodel exposes some parts that are specific to the refactoring domain. Having different types of collaborations (i.e., different kinds of role features) is not a common property of role modeling. Also, the modifiers for roles have been added specifically to model the data structures required to perform refactorings. For example, `optional` role types are not required to be bound. That is, one can model data structures which are not mandatory. Rather, additional functionality is provided by a tool—in this case a generic refactoring—if the optional role types are bound.

Using different kinds of collaborations (e.g., `RoleComposition` and `RoleAssociation`) can also be realized using a dedicated role feature instead of multiple subclasses. For example, one can define a role feature `collaborationType` that provides information about the kind of collaboration. Thus, the role modeling metamodel is quite similar to the metamodel presented in Sect. 8.4.1, even though it is not as minimal.

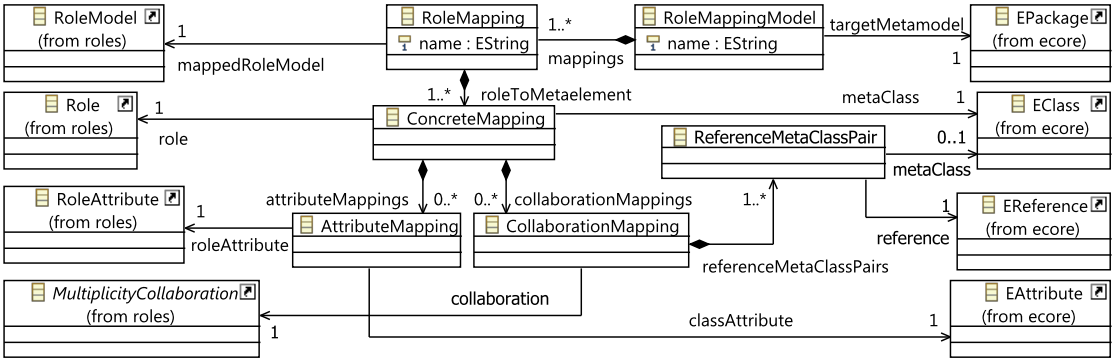


Figure 9.12: Refactory role mapping language metamodel.

Role Binding

In the case of Refactory we bind role types and role features (i.e., collaborations and attributes) to elements of metamodels. The former are bound to metaclasses, while the latter are mapped to references or attributes of metaclasses. To specify this binding a special role mapping language is provided as part of Refactory. The metamodel for this language is shown in Fig. 9.12.

One can find different types of mappings here. First, `ConcreteMapping`s connect a role type and a metaclass from the target language. Second, there are `AttributeMapping`s and `CollaborationMapping`s to bind role attributes to attributes from the metamodel and collaborations to potentially multiple references of the metamodel.

For Refactory we did not employ the generic role binding language that was proposed in Sect. 8.4.2, but rather a specialized language that is more suitable to the refactoring domain. The aim of this decision was to ease the mapping process for Refactory users as much as possible. Nonetheless, we are convinced that the same idea can be realized based on a generic role binding language.

9.4.3 Discussion

Applicability

Building a generic refactoring tool based on the principles of role-based RTE was a significant success. Creating role models of the data structures that are required by different refactorings is both easy and intuitive. Even though we did not technically reuse the languages from Chap. 8, the concepts and ideas of role-based tool integration account for the fact that Refactory became a very generic and valuable tool.

During the implementation of Refactory we customized both the role modeling and the role binding language. However, we consider the additionally introduced concepts

to be technical details which can also be realized on top of the minimal languages from Chap. 8. We are also deeply convinced that the idea of using role models to decouple tools from concrete data representations can be transferred to other application areas. Refactoring is only one example, where required flexibility can be realized this way.

Gained Benefits

By using role modeling in the context of model refactorings we were able to build a generic refactoring tool that can be applied to arbitrary modeling languages. Not only can we reuse the tool, which mainly consists of the interpreter for transformation specifications and some specification editors, but we can also specify and reuse generic model refactorings. Consequently, we require solely a mapping specification to enable a given refactoring for a new modeling language. This is a significant advance compared to other approaches where refactorings apply only to a particular language [224, 225] or a particular type of language [226]. Therefore, we were able to instantiate 47 concrete refactorings by mapping 6 generic ones to 16 metamodels [213].

Discovered Drawbacks

We have mentioned some of the downsides that are implied by the role-based tool integration approach in Sect. 8.5. One of the points mentioned there was the fact that tools need to be specifically designed to allow their interoperability. Even though Refactory was built from scratch, we experienced this drawback. For example, the specification of conditions that must be met by a model before a refactoring is applicable could be expressed with OCL constraints. However, since existing OCL tools are not implemented on top of role models, reusing these tools was not possible.

Furthermore, the realization of the Refactory role binding language posed some problems. As outlined in Sect. 8.4.2 one must either interpret role binding specifications or generate code that implements the semantics of the defined binding. For Refactory, we chose to use an interpreter. This interpreter—the one that realizes the role binding—is integrated with the transformation specification interpreter—the one that performs refactorings. Mixing up these two concerns resulted in complex interpreter code.

The probably most serious drawback that was discovered by the evaluation presented in this section is the dependency of our role-based tool integration approach to a concrete role modeling language. We have seen that we were induced to use a custom role modeling language for the implementation of Refactory. Even though, we have argued that the very same result can be achieved using a minimal role modeling language as presented in Chap. 8, this shows the need for a standardized language. Without such a standard even tools that employ role modeling conceptually cannot be easily integrated because they might use slightly different role modeling conceptualizations.

10 Related Work

Synchronizing related artifacts in software development has been a challenge for both researchers and practitioners for a long time. Consequently, numerous publications on the topic can be found in the literature. To provide a structured overview, we will classify existing work into the following areas.

First, publications that give a general introduction or definitions of RTE will be revised in Sect. 10.1. This does also include work where desired properties of RTE systems are specified, or goals of building RTE systems can be found. Second, we give an overview of approaches that target consistency checking and consistency management in Sect. 10.2, because these are important subtasks of RTE. Third, bidirectional transformations, being a popular means to address RTE scenarios are presented in Sect. 10.3. Fourth, adjacent problem areas are examined in Sect. 10.4, because their solutions may be transferable to model-based RTE. Fifth, concrete applications of RTE systems are presented in Sect. 10.5. These are very specific solutions for individual problems, but their results need to be compared to ours.

10.1 Definitions, Categorizations and Vision Statements

The term RTE can be informally defined as the *synchronization of related artifacts within software development processes*. To confine this quite general definition of RTE, several attempts have been made.

In [137] a definition of RTE is given that is based on the terms *view*, *view identification*, *view domain*, *view transformation* and *view operation*. A view is considered as a basic data container. The identification of a view is a function that can recognize valid views.

A view domain is the set of all valid views for a particular identification function. View transformations map views from one domain to another and view operations can alter a view in one particular domain.

While the definitions of [137] are certainly more concrete, the strong emphasis that is put on the term *view* can lead to some misunderstanding. Views, as defined in relational databases [227], contain less information than its original. We argue, that founding a definition of RTE on this premise seems too restrictive. In particular in MDSD one can find artifacts that require synchronization, but which are not a subset of each other.

A more generic, yet formal definition can be found in [136], where the relations between artifacts are defined as functions or *domain transformations*. Automation, being one of the ultimate goals of building RTE systems, is then defined by being able to mechanically derive inverse functions. If one can do so, data can be freely transformed between different domains. Moreover, if multiple domains are involved in a RTE scenario, certain architectural styles can be found. For example, domains can form chains to build *sequenced RTE systems*. Or, if domains represent other domains partially, *Model View Controller-based RTE systems* are obtained. Moreover, the term *Bidirectional Weaver (Beaver)* is used if domains are composed to form a new domain.

Some examples presented in this thesis are Beaver systems. In particular the evaluation scenarios for the PREP approach (cf. Sect. 9.1 and Sect. 9.2) are instances of this style. However, we must emphasize that PREP is more general than using inverse functions, because it explicitly supports non-injective functions, for which no inverses exist.

Also, role bindings, being a key component of the role-based tool integration approach (cf. Chap. 8), can be considered as domain transformations. The discussion about the required properties for such bindings (cf. Sect. 8.5) reflects the statements about deriving inverses from [136]. We suggest that a role binding language should allow for invertible operations only, which essentially aims at ensuring the automatic computation of inverse domain transformations.

In [120], RTE is defined to be substantially different from solely composing forward and reverse engineering techniques. Sendall and Küster argue that the main goal of RTE is to *reconcile* artifacts after changes have been made, rather than deriving new artifact versions, as done in forward and reverse engineering. Changes made to derived artifacts must be conserved. In [120] it is assumed that all artifacts are (artificially) represented as models. Based on this assumption the following subtasks of RTE are defined:

1. defining under what circumstances models are consistent or inconsistent,
2. deciding when inconsistent models should be made consistent again,
3. devising a plan for reconciliation according to the expectation of users, and
4. applying the devised plan by reconciling the models.

In [228] one can find a superset of these steps. In addition to the four steps from [120], the detection of overlaps between models, the diagnosis of inconsistencies and the tracking of findings are included. In the scope of this thesis we did not investigate on how to determine model overlaps. We assumed that knowledge about existing overlaps is present or derived in a preceding step. Interestingly, there are approaches to perform this detection based on logics [229] and ontologies [230], which could be integrated with our work in Chap. 7. Also, we subsume the other two steps—diagnosing inconsistencies and tracking findings—by the devision and application of the reconciliation plan.

Two approaches presented in this thesis implement the four steps from [120], albeit in different ways. The PREP approach (cf. Chap. 6) explicitly introduces the notion of a consistency fitness function to decide whether models are consistent. Inconsistencies are resolved as soon as they are detected and the plan for reconciliation corresponds to the propagation and replay step. The ontology-based synchronization approach (cf. Chap. 7) employs OWL and SWRL rules to check models for consistency. Reconciliation is performed according to the specialized algorithm presented in Sect. 7.5. Role-based tool integration, as discussed in Chap. 8, does not implement the four steps, since it avoids inconsistencies by design rather than performing reconciliation.

Besides the general steps that are required in RTE systems, Sendall and Küster identified a number of qualities that are generally desirable. These are:

1. the ability to manage trace information,
2. an intuitive and concise approach,
3. understanding the intention and expectation of users, and
4. assistance with detecting conflicts between model RTE activities.

The work in this thesis confirms these qualities. We have seen the crucial role of trace information in Chap. 6, as well as how important the intentions and expectations of users are. Moreover, we have identified that the latter can prevent full automation in some cases. If such situations arise (e.g., when there are multiple valid propagation options), we provide assistance by visualizing all options and the respective consequences of picking one of them.

In [144] one can find a more restrictive definition of RTE. The authors consider the synchronization of design diagrams and source code only. This can probably be explained by the time of publication where a lot of research was still focused on UML rather than on DSLs. Despite the fact that [144] employs a more restrictive definition of RTE than we do, the paper contains some interesting properties of RTE systems. First, it coincides with [120], stating that RTE exceeds plain combinations of forward and reverse engineering. Second, it emphasizes that tool interoperability involves RTE. In particular the second statement is confirmed by our role-based approach to tool integration.

In addition, the authors of [144] argue that UML does not provide adequate concepts to represent information about source code. Even though we agree in this concrete example, we are convinced that it is not required for all languages involved in a RTE scenario to cover all involved concepts. Rather, languages must be adequate w.r.t. their purpose and thus restricted in some sense. Languages that can represent all data involved in a RTE scenario are only required for view-based approaches, where derived artifacts are subsets of a large, integrated model.

In [189], the creation of trace information and the propagation of changes have been identified as key components to successfully employ model transformations. Our work confirms this observation and provides concrete methods to implement traceability and change propagation. The former is mainly used in Chap. 6, while the latter issues are equally addressed by Chap. 6, 7, and 8.

Since RTE aims at the synchronization of multiple related artifacts, the question to what extent the semantics of the involved data is preserved emerges. Kleppe and Warmer have shown that model transformations do not preserve semantics by definition [231]. Rather, one faces situations where semantics is not preserved or even cannot be preserved. The authors state that, at best, the semantics of the input language is expressed in terms of the output language. Since we did not incorporate formal descriptions of the semantics of the models that are subject to synchronization, we must share this point of view. Although we think that more investigation w.r.t. the preservation of semantics are required, our approaches cannot give such guarantees. We provide methods to ensure that input models are consistent w.r.t. their translated version in the target language, but the correctness of the translation itself is out of our scope.

Recently, another definition of RTE systems has been presented in [156]. It is also based on the idea of translating changes from target to source domains (cf. Chap. 6). Our approach was developed in parallel and is conceptually very close to [156]. Therefore, we base the definition of backpropagation-based RTE systems on [156]. Both Chap. 6 and 7 completely agree with the idea of change translation. However, our presentation of backpropagation-based RTE emphasizes practical feasibility. One can say that we have confirmed the formalization of [156] by the evaluation in Sect. 9.1 and 9.2.

A feature-model based formalization of the design space of programs dealing with heterogeneous synchronization—called *synchronizers*—is presented in [186]. First, the types of relations that can hold between different types of artifacts are analyzed. Here, one can find bijective (1 to 1), surjective (n to 1) and general (n to m) relations. Then, three main categories of synchronizers are defined—*unidirectional*, *bidirectional*, and *bidirectional with reconciliation*. The first category (unidirectional) covers synchronizers that synchronize target with source artifacts only. The second category (bidirectional) can be used to synchronize in both directions, but is only applicable if the source and target do not change both between two synchronizations. The last category (bidirectional with reconciliation) overcomes this limitation and can deal with situations where both target and source have been changed since the last synchronization.

Our work on backpropagation-based RTE is mainly settled in the second category. According to the definitions of [186], we compose a *unidirectional to-one artifact translator*—toward the target—with another *unidirectional to-many update translator*—toward the source—to form a *bidirectional synchronizer*. PREP can probably be extended to support concurrent editing of both source and target models, but this requires to analyze whether the propagated changes and the source edits are in conflict.

The ontology-based RTE approach from Chap. 7 is also bidirectional, but employs two *unidirectional to-one update translators*. The role-based tool integration (cf. Chap. 8) cannot be filed in one of the three categories, because the physical sharing of data, which is inherent to the approach, does not allow the distinction between category two and three. All changes are immediately propagated to the physical data representation. Thus, this approach is bidirectional and avoids simultaneous, potentially conflicting edits in the source and target by construction. At best, we can classify it as a composition of multiple *unidirectional to-one update translators*.

Antkiewicz and Czarnecki [186] explicitly mention that synchronizers may be interactive, if users must decide among multiple alternative options, which we did observe too. In Sect. 8.5 of [186] an enumeration of possible strategies for this selection process are given. During our evaluation, we employed *pre-determined choices* (in Sect. 9.1), as well as *interactive selection* (in Sect. 9.2). Furthermore, our approaches target *instantaneous synchronization* according to the definition of [186].

A categorization for bidirectional model synchronization approaches based on algebraic models has been presented in [232]. According to this classification, both our main approaches (i.e., PREP and role-based tool integration) are *incremental synchronization* techniques. We translate updates instead of whole models.

General properties of bidirectional transformations and RTE systems have also been presented in [87]. The first property—*correctness*—requires transformations to establish a consistent state when executing it in either direction. The second property—*hippocraticness* or *check-then-enforce*—states that neither the source nor the target model should be modified if they are already in a consistent state. The third property—*undoability*—requires that the application of a change followed by an immediate undo operation, yields exactly the pair of models that was present before. However, as already argued in [87], the last requirements is too strong to apply it to all bidirectional transformations. If a transformation meets all three requirements it is denoted as *coherent*.

10.2 Consistency Checking and Management

Early work on checking software artifacts for consistency has been performed in the context of requirements engineering. Here, different stakeholders can disagree over the domain that is subject to software design. Often, the term *view points* is used to denote the different perspectives people hold [233]. To resolve conflicts between view points,

research on computer-supported negotiation has been conducted about twenty years ago [234]. Back then, the focus was on reconciling domain models that are developed by groups of different people. The work in this thesis is different as we assume consensus among developers about the domain, but rather focus on conflicts (i.e., inconsistencies) that can appear even if only one developer is involved. Thus, we focus on inconsistencies that are caused by the complex relations between artifacts instead of disagreement w.r.t. the modeled domain.

Despite the somewhat different motivation of our work, checking models for consistency is an essential subtask of RTE [5, 120]. Even though there are arguments that inconsistencies must be tolerated in software engineering (e.g., [235]), this merely refers to the fact that there are temporary situations in development where further actions are required to restore consistency.

In general, one can divide consistency checking into two categories. First, there are checks for individual models. Second, checks across multiple, related models are required. The former category can be implemented using a constraint language (e.g., OCL). However, checking large models (e.g., instances of the UML) can require huge effort if models and sets of constraints grow large.

For example, in [236] checking large UML models for consistency is addressed by incremental constraint checking. Once inconsistencies have been detected, fixes are required. Solutions to this have been presented in [237], where choices for fixes are automatically derived. Again, choices must be picked by the developer and tool support is only given to provide assistance. In comparison to this thesis, the work of Egyed [236, 237, 238] is specific to UML design models. Also, Egyed does restore consistency within individual models only, but does not incorporate consistency across models as we do. However, we share the position that providing developers with a valid set of choices for restoring consistency in required in some cases. One can say, that our work partially transfers concepts from Egyed's work to multiple models. Our focus is put on global consistency rather than ensuring local constraints.

To check whether models are globally consistent, the relations between the respective metamodels need to be specified. Interestingly, the integration of metamodels has also been investigated by the requirements engineering community [239]. Albeit we have more sophisticated modeling standards and environments today, the idea of language integration to ensure consistency can be traced back to the late nineties.

A more recent method for model merging is presented in [117], where homogeneous models are merged to form a single model, and global consistency constraints can be checked in the same way local ones are evaluated. In [118] Diskin et al. generalized the work of [117] and presented a method to define consistency across heterogeneous models. Diskin and his colleagues propose to merge the involved metamodels, but only to *an unavoidable minimum*. Likewise, the *relevant parts* of models are merged.

Both approaches [117, 118] are structural consistency checks in the sense that they do not take the semantics of the involved models into account. If one wants to include this,

approaches that map models to a common semantic domain are required. Examples for such mappings can be found in [240, 241, 242].

To illustrate the idea, we would like to briefly mention [240], where the consistency of object-oriented behavioral models has been investigated. The authors present a procedure to check that models are consistent w.r.t. their semantics. To do so, models are translated to a semantic domain—a formalism with well defined meaning (e.g., Communicating Sequential Processes (CSP) [243]). While the proposed approach allows to precisely check global consistency, it does not provide means to restore consistency automatically. Therefore, is can be combined with our approaches from Chap. 6 and 7 to check models for consistency, but it does not provide inconsistency resolution itself.

Other examples for checking semantic relations between software artifacts can be found in [244, 245]. Here, requirement specifications are mapped to formal logics—the common semantic domain—to check whether and how these specifications overlap. This is similar to our work on ontology-based synchronization specifications, but restricted to the detection of overlaps and inconsistencies. We extend this work by providing actions that resolve inconsistent states. A similar extension can be found in [229], where Hunter and Nuseibeh introduce a special logic variant that allows to represent inconsistent knowledge. We do not require such support for inconsistency, because we model elements that are required to restore consistency as unknown facts. This is supported by the OWA, which is a central property of OWL.

A special use case of Consistency Management (CM) can be found in [246]. The authors formally define UML class diagram-like structures as graphs and show how consistency between views on theses graphs can be preserved. Consistency is formalized as constraints over multiplicities of edges. Using a train control system as case study the authors show a constraint solving algorithm can make sure that the different views satisfy the formulated consistency constraints beforehand. However, no procedure is presented that allows to restore consistency if it is broken.

To ensure consistency between concurrently modified artifacts, one can also employ transactions, as known from databases. In [247] multiple users can edit dependent diagrams and synchronize their modifications using a transaction mechanism. As in [246], consistency constraints define the dependencies between related objects. Unfortunately transactions and consistency checks alone cannot repair violated consistency constraints. Furthermore, even if transactions are used, one should make sure that interleaved steps within a single transaction produce the same result independently of their order.

In addition to the approaches based on model merging, mapping to semantic domains, and transactions, arbitrary *black-box consistency checks* can be employed in RTE. However, one must be aware of the fact, the solely checking consistency may not help in restoring consistent states.

Consistency checking is only part of a RTE system, because restoring a consistent state when inconsistencies are detected is equally important and often comparatively harder. For the work presented in this thesis, checking consistency was not the main

focus. The backpropagation-based approach to RTE defines consistent states as a direct consequence of replaying the forward transformation. Thus, no explicit consistency check was required. Also, role-based tool integration propagates all changes directly to the physically shared data, which implies that inconsistencies cannot arise at all.

To address inconsistencies (i.e., *to manage them*), Easterbrook and Nuseibeh introduced a framework based on the notion of *view points* [248]. These view points are similar to the role models introduced in Chap. 8 as they provide specific abstractions that are required for different development activities. However, the framework is restricted to maintain information about the relations between different view points. Thus, the focus is rather on providing developers with information that can be useful when resolving inconsistencies, but no automatic procedures are provided to do so.

In [249], Sabetzadeh continued the work of Easterbrook and presented a high-level, formal framework for managing inconsistent and incomplete views. The framework is based on category theory and defines conceptual patterns to derive views as well as operators that are essential to manage views. Early results of implementing the framework have been reported in [250]. In general, one can observe that views in [249], are very loosely connected with each other. The framework and its implementation employs mappings between views, but these do not need to correspond to functions that derive views automatically. This is in contrast to our work, where derived views on models are strongly coupled to the original model by means of a model transformation or role binding. Also, the report on the framework's implementation [250] mentions that the integration of category theory is subject to future work. Thus, evidence whether practical views on models can be realized on this theoretical basis is still missing.

More recent approaches to inconsistency management [251] do not only detect inconsistencies, but also try to assist developers to resolve conflicts. In [251], inconsistencies in UML models are detected by the application of rules specified in AGG. If a rule matches a specified pattern (e.g., an abstract operation is defined in a concrete UML class), a new model element `Conflict` is created. To resolve detected inconsistencies, a set of *resolution rules* must be specified. For the given example, making the abstract operation a concrete one is such a resolution rule. In contrast to our work, which focuses on cross-model consistency constraints, which are caused by the derivation of models using transformations, the approach in [251] targets constraints within single models. Thus, it can complement our work with support for intra-model consistency management.

10.3 Bidirectional Transformations

10.3.1 QVTR and Triple Graph Grammars

Bidirectional model transformations are among the most popular candidates to solve RTE problems. For example, QVTR and the TGG formalism are put forward in this context. The latter approach has been presented in 1994 by Andy Schürr [79]. It can

be considered one of the first and also one of the most mature formal approaches to bidirectional transformation. It has also heavily influenced the design of QVTR, which is now an OMG standard [83], but also significantly younger. A detailed comparison of the QVT and TGG formalisms can be found in [84].

Both formalisms are based on the idea of direction-independent transformation rules. Rule designers state the relations that must hold between source and target models, instead of differentiating between conditions that must hold before a rule is applied and the conditions that will hold after a rule was applied. To actually transform models in one direction or the other (i.e., either forward or backward), direction-specific rules are derived from direction-independent ones. Both forward and backward rules are derived from a single specification. In addition to their support for bidirectional transformations, the particular strengths of QVTR and TGG-based tools are typed traceability links and, in the case of TGGs, their formal semantics.

In addition to the bidirectional execution of TGG rules, support for incremental updates has been investigated by Giese and Wagner in [252]. This work was both motivated by the need to support synchronization scenarios, as well as the potential to speed up transformations. Giese and Wagner also observed that transformations can be non-injective, which yields multiple valid source models for a single target model. However, the approach presented in [252] is restricted to the bijective case. Our work on backpropagation-based RTE does not expose this limitation.

Perdita Stevens analyzed the QVT standard w.r.t. its support for bidirectional transformations [87]. She explicitly distinguishes between bijective and bidirectional transformations, the former being a subset of the latter. Stevens holds the view that MDSD demands for transformations that are bidirectional, but not bijective. She argues that the refinement of models toward system implementations requires the addition of details (e.g., about the target platform or the domain itself). We completely agree with this. However, we found the argument that requiring target models to contain more information than source models in MDSD is not the sole cause for non-bijectivity. We do even argue that target models must not contain more information, but bijectivity is still insufficient. The idea of refinement by composition heavily employs transformations that are not injective, but also yields target models that cannot contain facts which are not present in one of the source models.

In [253, 254] Jakob et al. introduce a special kind of TGG rules, called *VTGG* rules, that can be used to define non-materialized views on models. VTGG rules are restricted in the sense that they allow to create only one element on the view side of each rule. This allows to translate view modifications more easily to the respective base model. Also, the correspondence part of the TGG rules is replaced by plain links carrying a special `new` tag. To implement non-materialized views that are defined by VTGG rules, the authors propose to use the class adapter design pattern [50].

VTGG rules are closely related to our role-based approach to tool integration presented in Chap. 8. One can consider VTGG rules as a role binding language. Since VTGG

rules are restricted (e.g., only one element can be created on the view side of a rule), not all tool integration scenarios can be handled. We believe that from practical point of view it is required to have a role binding language that provides both concepts that preserve bidirectionality and others which do not. The use of role models, as presented in Chap. 8, allows both to retrieve data from existing models—to create views, but one can also augment existing domain abstractions with new information. Moreover, the choice about which parts of models are materialized and which are virtual, is left to the tool integrator in our approach. Thus, one is not tied to existing decisions w.r.t. the materialization of data. Our virtualization is dynamic.

In [255] Ehrig et al. proof that a restricted subset of TGG rules—all TGG rules which are non-deleting—preserve information (i.e., they are bijective). While this gives a great guarantee if one employs such TGG rules, applying the TGG formalism to scenarios with non-injective relations is not possible.

In summary one can say that QVTR and the TGG formalism are both suitable to build a certain class of RTE systems, which was confirmed by practical applications [256, 257]. However, there is also a need for methods that deal with unidirectional and non-injective transformations, because of their widespread practical use.

10.3.2 Bidirectional Lenses

Relational lenses have been introduced by Benjamin C. Pierce and his colleagues [155, 190]. Pierce defines a lens as a pair of functions, where one function (`get`) extracts an abstract view from some data structure and the other function (`put`) is used to translate the abstract view back to the original representation. To permit views that are actual abstractions (i.e., views that contain less information than their originals), the previous state of the original is provided to the `put` function when translating views back to originals. Lenses have been applied to synchronize tree-structured data [258], the database view update problem [155, 190] (cf. Sect. 10.4.1), and string data [148].

In [143] Benjamin Pierce' work on lenses was recapitulated and rephrased using algebraic theory (i.e., relations between sets of models and functions on models). It turned out that the definitions of lenses and the respective laws fit very smoothly into the algebraic framework. This suggests that the laws of lenses can be generalized and are applicable to other to bidirectional transformation approaches.

Lenses are based on forward transformations that are functions on the source model only. Therefore, target models can be abstractions of the source model only [87]. The target model must contain less information. This is a restriction that does also apply to our backpropagation-based approach to RTE and role-based tool integration, but not to the ontology-based synchronization method. Moreover, lenses require injective relations between models. The `put` function returns a single source artifact. Since we are convinced that non-injective transformations are frequently required, we consider this as a drawback, which is not exposed by backpropagation-based RTE.

Very recently, Diskin et al. have shown that state-based bidirectional model transformations are a special case of delta-based ones [259]. The authors show how to generally construct a state-based system from a delta-based system. They also explain why the opposite is not as easy. Based on the Pierce's lenses [155, 190], which are state-based, a delta-based extension called *u-lenses* is presented. Our PREP approach can be considered to be partially delta-based, because we translate updates from the target domain to the source domain, but execute the replay step on the whole source model.

10.3.3 Others

Another approach to bidirectional model transformation can be found in [260] (and more detailed in [261, 262]). The authors propose a Bidirectional Object-oriented Transformation Language (BOTL). Rules in BOTL consist of two parts. The first (upper) part is a pattern defined in terms of the source metamodel. The second (lower) part is a fragment containing elements of the target metamodel. This scheme is similar a TGG rule, but omits the correspondence graph. During the translation a target fragment is created for each match found in the source model. If multiple rules create target elements, the results are merged by name. This merge operation implies a relation between source and target elements that can be considered as implicit correspondence graph.

To establish bidirectional transformations, rules must adhere to a number of restrictions, which can be found in [262]. These restrictions enforce that each rule is bijective and that each type of target elements is created by one rule only. Thus, the restrictions ease the computation of inverse transformations, but do also significantly reduce the expressiveness of BOTL. Also, no implementation of BOTL is available. The proof of practical applicability of the theoretical concepts is still missing.

In [263] Sendall proposed a transformation language *Gmorph* which combines the concepts of graph rewriting and generative programming [216]. Even though this language could potentially be employed for bidirectional transformations, no further work can be found. The question to what extent Gmorph can support RTE remains unanswered.

Burbeck and Larsson [264, 265] have investigated on the inversion of different rewrite approaches—lambda calculus, combinatory logic, term rewrite systems, attribute grammars with forwarding, as well as graph rewriting with the DPO and SPO approach. They concluded that the Lambda Calculus and logic are too low level and too difficult to invert. Term rewrite systems can be inverted if they adhere to some restrictions (no variables that appear on the left hand side of productions only). Attribute Grammars with Forwarding were discarded because their productions cannot be easily inverted. The remaining two approaches (DPO and SPO) showed that only the first one provides invertible rules, because the latter does not enforce the *gluing condition*. *CODEX*—the system developed in [265]—cannot handle one-to-many and many-to-many mappings, because of the identification condition of the DPO approach. They also conclude that one cannot delete arbitrary model elements because of the dangling edges condition.

In [266], an approach close to PREP is presented. Here, Xiong et al. propagate changes that are applied to target models of ATL transformations to respective source models. The concrete translation of a change is performed by considering the byte code of the ATL transformation. Thus, the idea is similar to PREP, but the concrete approach exposes some several limitations.

First, insertions in target models cannot be handled. This restriction does not apply to PREP. Second, the translation of changes assumes injective ATL transformations. Our evaluation has shown that this property does not hold for practical MDSD processes. Nonetheless, we think that [266] confirms several key ideas of PREP, for example, preferring the translation of individual changes over the inverse transformation of whole models. Moreover, we consider [266] as a basis for further investigation w.r.t. the translation of changes in heterogeneous scenarios. ATL is a widespread used transformation language and establishing support for RTE in this context is highly relevant.

Later on, Xiong et at. presented an more generic approach to synchronize software artifacts [267]. Again, changes are translated in order to synchronize models, which is similar to our work. The specification of this translation is encapsulated in so-called *synchronizers*, for which certain desirable properties can be shown. The main difference w.r.t. PREP is that we define the translation of changes independent of concrete relations between artifacts. Rather, we refer to atomic transformations operators. Thus, we can reuse the same translation rules for multiple transformations, assuming the same transformation operators are used.

10.4 Similar Problem Areas

10.4.1 Database View Update Problem

In relational database systems a problem called *view update problem* arises [138, 139]. Whenever a view on one or more database tables is defined, changes made to this view must be propagated to the respective source relations. However, in general it is not clear which tuples in the source relation are affected by changes in the view and how changes can be translated. There can be different possible backward mappings, as well as changes that cannot be propagated to the original tables at all.

In [139] a seminal solution to the problem was presented. It is based on the idea that changes applied to views must be translated to changes in the underlying tables. Motivated by the specific problem context, changes are modeled according to the domain of relational structures. Also, views are defined based on relational query languages.

Dayal and Bernstein emphasize the need for a *semantic integrity constraint* in [139], which is similar to our global consistency constraints. The approach presented in [138] is similar, but according to [268] of less practical use. Dayal's work is groundbreaking as it contains many ideas that we have reused and extended to support RTE in MDSD.

The work on the view update problem has been continued in the database community [140, 141, 269] and some of the general ideas have significantly influenced the work on model synchronization and RTE. Most of the publications that directly motivated this thesis (e.g., [107, 147, 156]), do apply similar principles. However, one must say that even though the idea of change propagation goes back to the early eighties, transferring it to models and modeling languages was far from trivial.

Dayal's work is restricted to relational data models and relational query languages. We consider our work to be more general since we support models that are based on EMOF (i.e., typed, attributed graphs) and other transformation languages. Also, the representational issues that are addressed by our bridges between technical spaces, are not examined in [139], because Dayal's work is conducted within a single technical space—relational database systems.

Years later, the view update problem was revisited and tackled using a custom bidirectional query language [190]. Every expression in this language does not only state a view definition (as in usual relational query languages such as SQL), but additionally provides a view update policy. This policy is used to decide how updates to a derived view are handled. This approach (explicit existence of information about backward transformation) is therefore a generalization and might be applicable in other scenarios too. Another interesting result shown in [190] is that basic operators (projection, selection and join) can be composed preserving the bidirectionality.

10.4.2 Model Versioning, Metamodel Evolution and Extensibility

In [270] an algebraic framework to model versioning is presented. Model versioning is concerned with multiple replicas of the same model—at least one for each developer—and the evolution of individual models—caused by the application of changes. The authors introduce the *tile notation*, which comprises a model, a replica of this model, as well as two evolved versions of the former two. These four models form a *tile*. Based on this, *tile systems*—compositions of multiple tiles—can be constructed. These systems can be mapped to *double categories* (i.e., categories of categories)—an algebraic structure that is part of category theory (cf. [271] for an introduction).

If one can show that the basic update operations on models adhere to the requirements of establishing a category—if they can be considered as arrows—results from category theory can be reused. For example, in [270] the *pasting lemma* states that any composition of compatible tiles yields the same result. However, the proof that model edits fit the prerequisites of category theory is still missing.

Although our work does not primarily target model versioning scenarios, we are convinced that PREP can be applied here, at least to a certain extent. Model versioning heavily involves the creation of copies of models, which allows to propagate changes comparatively easy. However, the distribution of models that is also an essential property of model versioning scenarios, requires special care.

A problem that is related to model versioning is *metamodel evolution*. If metamodels evolve over time, their instances need to evolve as well. In [272, 273] recent approaches to metamodel evolution can be found. In general, changes that are applied to metamodels can be coupled with respective transformation specifications to migrate existing models to a new version of their metamodel. We consider this as a special kind of change propagation. However, the translation of changes must be provided by language developers, because metamodel changes can be language-specific.

In this thesis, we did not consider the translation of changes across metalevels. But, the fact that both [272] and [273] rely on metamodel changes rather than on metamodel versions further confirms the significance of find-grained information about changes. We make use of the same idea, albeit solely within one metalevel at a time.

In [144], the extensibility of modeling languages in general and of UML in particular is considered as a prerequisite to enable RTE. The authors describe a scenario, where a metrics tool, a program visualization tool, and a refactoring tool shall be used and operate on the same system specification, but use appropriate representations of the program according to the special needs of each individual tool. This poses a problem, because essential concepts from the source code (e.g., method invocations and attribute accesses) are not available in the UML metamodel. Also, these concepts cannot be easily integrated using the extension mechanisms provided by UML.

A practical approach to metamodel extension has been presented in [205]. Here, metamodels can be merged with *extensions* to obtain integrated metamodels that provide both the concepts of the original and the extension. The example in [205] relies on merging by name, but other identification criteria are equally valid. If tools are developed independently the approach is not applicable, because metamodel extensions depend on the initial metamodel.

In the context of RTE, the limitations w.r.t. the extensibility of metamodels are highly relevant. First, if metamodels can be smoothly integrated, tools can operate on compatible domain abstractions and redundancy can be avoided. Second, if concepts cannot be expressed in a modeling language, changes cannot be propagated as desired. One is limited by the expressiveness of the involved modeling languages and cannot freely introduce extensions that may be required by a particular RTE scenario. Our role-based tool integration approach explicitly targets such situations and resolves the limitations of metamodel extensibility by introducing more flexible bindings between metamodels.

10.4.3 Refactoring

Refactorings have been defined by William F. Opdyke as *restructuring operations that preserve the behavior of programs* [222]. Refactorings are similar to model synchronization, because changes that are applied while refactoring a program, do often require corresponding changes in order to keep the overall behavior identical. Refactorings are different from model synchronization, because the changes that users apply to models

can deliberately alter the behavior of the modeled system, whereas refactorings must not do so. Thus, both aim to preserve different kinds of constraints.

In [274] a variety of refactorings for UML have been presented, extending the classical application area of refactorings from programming languages to a widespread modeling language. More recent approaches do even target arbitrary modeling languages [226], including our role-based refactoring approach [213].

These publications show that refactorings can be implemented by model transformations. But, they also clearly indicate that refactorings are specified manually. Each refactoring requires a custom transformation that preserves desired properties (e.g., the dynamic semantics of the models). Even though, the same kind of change is applied (e.g., a model element is moved), different transformations are required. For example, pulling up a feature in a class diagram is a substantially different from pulling up a method. The language-specific semantics demands for varying transformation specifications.

If one wants to employ PREP to realize model refactorings, multiple aspects need to be considered. First, the synchronization fitness functions must be designed such that the preservation of semantics is captured. That is, only propagation options that result in semantically equivalent models must be accepted. Second, the translation of changes is language-specific. For example, instead of handling additions of arbitrary model elements the same, the types of these elements (i.e., their metaclasses) are relevant when translating changes. We think that refactorings can be realized within PREP, but it is not clear whether the particular strengths of our approach are beneficial in this context. The support for non-injective transformations is probably not required, because refactorings usually yield only one valid way to adjust corresponding parts of a model. However, definite answers require future investigations.

A particular application of refactorings in the context of framework evolution has been presented in [275]. Here, refactorings were employed to capture the changes of framework APIs. Based on this information, inverse refactorings—so-called *comebacks*—allowed to adapt plug-ins that were compiled against a previous version of the framework to new versions. One can say that changes applied to the framework API are reflected by applying their inverses to plug-in adapters. The constraint that is preserved by this procedure is the compatibility of the plug-ins w.r.t. their linking against the framework. Also, the approach aims at the preservation of the dynamic semantics. Old plug-ins must behave correctly even in the presence of a new framework version.

We consider [275] as a very special case of change translation. API changes are translated to plug-in adapter changes. Also, custom constraints w.r.t. the preservation of the static linking and the behavior apply. We think that the work on framework evolution can be generalized to arbitrary synchronization scenarios. In principle, the approach is similar to PREP (i.e., changes are translated and constraints must be satisfied to ensure the correctness of the overall transformation), but the language-specific nature of refactorings may eliminate the benefits of PREP. For example, the translation of changes according to the primitive change operations is not possible anymore.

Additionally, our work on role-based tool integration could provide an alternative solution to the framework evolution problem. Instead of generating adapters to connect old plug-ins with new framework versions, role bindings could be employed in this context.

10.4.4 Tool Integration and Modeling Platforms

The need for a posteriori tool integration has motivated [185] and the authors propose to build *wrappers* to provide homogeneous access to data involved in such scenarios. These wrappers bridge the technical quirks of individual tools and can therefore be considered a small-scale variant of our bridges for technical spaces (cf. Chap. 5). Due to their specific nature, wrappers are less reusable, but still required if proprietary tools do not conform to the concepts of a particular technical space.

Based on the technical integration provided by the wrappers, Becker and his colleagues employed transformation rules to propagate changes between tools. The rules in [185] are inspired by the TGG formalism, but modified to meet the requirement of the practical tool integration problem. Similar to using general TGG rules, the authors discovered problems w.r.t. rule execution order and rule overlaps. Consequently, an algorithm is presented that tries to minimize the number of situations in which such conflict arise.

Even though, we do also target tool integration in Chap. 8, our approach is significantly different. We explicitly anticipate tool integration during the design of tools, whereas Becker et al. had to deal with isolated tools that were probably not anticipated to be used together. We do believe that a lot of effort that was required to achieve the results of [185] can be avoided in the future if tool integration becomes a primary design goal. We have presented an approach to achieve this goal with small effort.

In [276], tool integration based on *modeling services* is presented. By invoking such services, tools are allowed to alter models other tools operate one. The connection that is established between tools in this fashion is similar to our role-based tool integration approach, but we explicitly separate the integration aspect from the tools. Thus, we do not let tools directly access the models of other tools. Rather, each tool operates on its appropriate domain abstraction—its role model. As a consequence, we can flexibly change the integration of tools, whereas the approach in [276] requires tool modifications in this case, as the invocation of services needs to be altered.

Jossic et al. [277] propose to use weaving models and model transformations to integrate models. Based a heuristic weaving that states the relation between different metamodels, transformations derive models from each other. In contrast to PREP, which is a change-based approach, the transformations in [277] process whole models and are therefore state-based. Also, the derived transformations replicate data, which is not the case for role-based tool integration. Nonetheless, the discovery algorithms presented in [277] could be used to generate default role bindings.

In [278] the concept of a *virtual tool* was introduced to allow the model-based integration of concrete tools. Based on a configuration, metamodels of tools are combined

and adapters are used to derived data from existing tools. Our role-based approach to tool integration is quite similar in the sense that we use role bindings to configure the relations between metamodels. Also, we can provide virtual views on existing domain abstractions. But, in contrast to [278], we share data physically between tools instead of using transformations to switch between different representations. Also, we allow tool integrators to choose freely about which data is derived and which is stored physically.

A more general view on model-based tool integration can be found in [279]. Here, Bézivin et al. emphasize the role of metamodels during tool specification, the role of model transformations and mappings for tool integration, as well as the need for extractors and injectors to obtain models from tool-specific data. We completely agree with [279] as it is along our argument to separate technical from logical synchronization. Moreover, even though we propose the use of role models during tool design, we pursue the same objectives. That is, we want to decouple the domain abstraction required by a tool from concrete repositories. Our evaluation in Sect. 9.4 followed the idea of building an *ideal tool* [279] and confirmed the benefits that can be gained by this procedure.

Recently, Broy et al. [280] reported on the tool integration issues in the automotive and avionic industries. They state that current engineering environments consist of ad-hoc composed tool chains. Broy et al. focus on high-level requirements that must be met to obtain seamless environments. We think that particularly the bridges for technological spaces and the role-based tool integration can provide substantial support in this context. Although we cannot address all requirements, our approaches certainly target some important ones (e.g., the integration of data repositories).

A concrete modeling platform and toolbox as requested by [280] is the AMMA platform [281]. AMMA consists of the Kernel Modeling Framework (KMF) that provides low-level operations on models, the Common Modeling Runtime (CMR) which is a virtual machine that can execute KMF operations, a set of projectors to bridge other technical spaces and Global Model Management (GMM) to manage relations between models. The latter allows the creation of megamodels (cf. Sect. 2.1.4) for which a case study can be found in [282]. The bridges presented in Chap. 5 follow the principles of AMMA (e.g., using projectors to inject software artifacts). We are convinced that PREP can be realized on top of KMF and AMMA. Also, we think that the integration of role modeling into AMMA could increase the potential w.r.t. the integration of metamodels.

10.5 Concrete RTE Applications

The literature exposes various examples of specialized RTE approaches. For instance, Paesschen and Hondt [283] investigate RTE for entity relationship diagrams and their implementation in the language Self [284]. The authors keep diagrams and the respective implementation in sync, by using the same physical objects to represent both. This is similar to our role-based tool integration approach, but specific to this application.

Similar to role-based tool integration, there is no explicit synchronization process, because all change operations operate on the same physical data. Changes are immediately reflected to both views (the Self code and the diagrams). However, the approach is sketched very roughly. It is not clear how changes made in diagrams are propagated to the underlying data objects. Furthermore, there is no information how the Self source code and the diagrams are derived from the common set of elements. One could interpret this approach as dealing with one domain only. The source code and the diagrams are only different visual representations of in-memory objects. The translation of changes is easy in this case. A respective role binding would probably be very simple.

Chalabine and Kessler [147] presented a RTE system for a specialized operator—the *double grafting operator*—which can be used for static aspect weaving. The authors do not have an implementation yet, but the paper shows two interesting results. First, they have observed that it is not always necessary to invert transformations. Sometimes it is sufficient if one can replay them. Second, depending on the transformation operator, the information required to determine the origin of a particular part of an artifact can be very small. We consider our work on backpropagation-based RTE as a generalization of [147], because we support transformations on graphs rather than on trees. Also, our approach is not limited to the double grafting operator.

Antkiewicz and Czarnecki [149] investigated on the synchronization of source code of Eclipse plug-ins and domain-specific models. The latter conform to a Framework-Specific Modeling Language (FSML) that allows to describe the interaction of workbench parts in Eclipse. Constraints on the FSML models ensure that framework extensions complete the framework in a proper way. The approach presented in [149] is called *agile RTE* and incorporates automatic forward engineering and partial (manual) reverse engineering. The latter includes a comparison of the newly extracted model (from the code) with the last known model used in the forward engineering process. Based on the result of this comparison, *consistent changes* are propagated (to the code or the FSML model). *Inconsistent changes* indicate conflicts (e.g., if the source code and the model have been changed in a contradictory way). Thus, resolving them requires manual effort.

The method presented in [149] transforms a heterogeneous synchronization scenario to a homogeneous one by reverse engineering FSML models from the source code. Then, the actual synchronization is performed by means of a three-way merge of the previous FSML model, the newly extracted one, and the asserted one. By analyzing the changes made on the code side and the ones applied to the asserted model, the distinction between consistent changes and inconsistent ones is made. The former can be applied automatically, while the latter need a manual decision to restore a consistent state.

The extraction of FSML models from code corresponds to our notion of extracting a skeleton (cf. Sect. 4). The fact that this skeleton is preserved to a large extent, even if changes are applied to the source code or the asserted model, allows to propagate changes as presented in [149]. The three-way merge algorithm relies exactly on the preservation of the skeleton. Consequently, we consider [149] to confirm our conceptual framework.

Another class of RTE methods that is particularly interesting, are methods to synchronize generated code and models. Few MDSD processes can completely do without manual source code extensions and thus, RTE issues arise quickly. In [146] a comparison of existing RTE tools for source code generation can be found. Furthermore, the authors propose their own method, which is based on three-way AST merging and tracing between models and code. According to [146], the proposed method allows editing source code at statement level without any need for protected regions. The details of the three-way merge algorithm can be found in [145].

The work presented in [146] confirms some of our key principles from Chap. 4. First, separating the extraction of the AST and the actual synchronization between models and the AST, follows our idea of strictly separating different synchronization concerns. Second, the authors employ trace links to keep models and ASTs in sync. Third, the reconciliation procedure is based on a three-way merge algorithm that relies on the preservation of a skeleton. Angyal et al. do even use the exact term to denote this. Thus, we consider [146] as an instance of our conceptual framework.

In [99] support for RTE in Fujaba was investigated. To reconstruct Fujaba Story Diagrams [96] from existing Java code, a parser and a custom back-end that creates diagram elements from the syntax tree is employed. This back-end uses annotations that are added stepwise. Different annotation engines detect patterns in the syntax tree (e.g., special method names) and add respective annotation elements to the tree. Subsequent to this process, Fujaba tries to extract graph rewriting rules that resemble the found annotations. These graph rewriting rules correspond to story diagrams and form the result of the re-engineering process.

This method has several drawbacks. First, the parser back-end must be written by hand. The recognition of the design (i.e., the Story Diagrams) is therefore error prone. As a consequence, there is no guarantee that an re-engineered diagram results in the same code when code is re-generated. Second, the approach is strongly tied to the Fujaba code generation algorithms and to Java in general. The parser back-end requires detailed knowledge about the way code was generated in the first place. As the parser (and the back-end) is specific to the Java language it cannot be reused for other programming languages. The RTE support is hard-coded. Third, the proposed RTE method cannot handle parts in the code that are not reflected in diagrams. Fujaba tries to avoid this problem by encoding method bodies in its story diagrams. But, code elements that cannot be detected by the reverse engineering algorithm are lost after regenerating code.

In [285] a rather ad-hoc approach to RTE is presented. The relationship between models and code is not explicitly modeled. Rather, various pragmatic ideas are used to extract models from source code or create source code from models. Both static and dynamic analysis is employed to gather information from existing source code. The synchronization seems hand-coded and it not declared formally. Thus, reasoning about the properties preserved or destroyed by the synchronization is not possible.

11 Summary, Conclusion and Future Work

11.1 Summary

In this thesis, we have presented a conceptual framework for the design of RTE systems (Chap. 4). The framework delivers definitions for essential terms in the context of RTE and a process that can guide the creation of new systems. We have argued that this process involves building bridges between technical spaces and analyzed concrete bridges to transfer heterogeneous artifacts to a common space (Chap. 5). To keep information logically synchronized, two novel approaches to RTE (Chap. 6 and 8) have been presented. Both approaches instantiate our conceptual framework, albeit in a different way. To ease the specification of synchronization rules, we have investigated on the feasibility of semantic web technologies for this purpose in Chap. 7. We will now shortly recapitulate the main properties of our work.

Our conceptual framework for RTE (Chap. 4) is based on the idea that technical and logical synchronization aspects should be cleanly separated. We emphasize that artifacts that are subject to synchronization need to be transferred to a common technical space before their contents can be aligned logically. Moreover, the framework puts emphasis on the partitioning of models into parts that require synchronization (skeletons) and parts that do not (clothings). Depending on the relations that hold between skeletons, one can determine whether resulting RTE systems may require manual decisions or not.

Bridges for technical spaces strive to provide the same kind of information using a different representational paradigm, ideally in an information preserving way. Consequently, conversions of artifacts between different technical spaces (i.e., bridges) should be lossless. Bridges can be established by replication (i.e., by providing a physical rep-

resentation of data in the opposite space) or adaptation (i.e., by implementation of the data access interface or query language of the opposite space).

Our backpropagation-based approach to RTE—the PREP approach—supports non-injective transformations where target models contain at most as much information as the source model. Thus, any change made to a target model can be propagated to the source model, possibly in multiple ways. Moreover, PREP is most feasible for compositional transformations, even though it is not limited to this case. PREP is an a posteriori approach and can thus be used with existing tools.

Role-based RTE allows to separate domain abstractions (i.e., views or role models) for individual tools from concrete data representations. Using role bindings, tools can be flexibly integrated. The translation between different representations is performed on the fly, no data is replicated. Heterogeneity is explicitly captured in this approach be using role models to encapsulate appropriate domain abstractions. Role-based RTE is an a priori approach and does therefore require special actions at tool design time.

Ontology-based RTE was motivated by the logical foundation of semantic web technologies. It can handle scenarios, where source and target models differ w.r.t. the amount of contained information. Thus, a target model can contain additional data, that cannot be derived from the respective source model. In addition, ontology-based RTE explicitly targets transformations between heterogeneous domains that contain physically replicated data.

The main contributions of this thesis can be summarized as follows:

- a conceptual framework for RTE defining common terms, definitions and a design process for RTE systems (Chap. 4),
- an a posteriori approach for change-propagation based model synchronization that supports non-injective transformations—PREP (Chap. 6), and
- an a priori method to anticipate arbitrary RTE scenarios at tools design time—role-based tool integration (Chap. 8).

Minor contributions are:

- a detailed analysis of bridges between technical spaces (Chap. 5), and
- a synchronization specification mechanism based on OWL and SWRL (Chap. 7).

11.2 Conclusion

In this thesis, we have analyzed different aspects that render the design of RTE systems difficult. For example, the fact that artifacts reside in heterogeneous technical spaces, the missing anticipation for tool integration, the lack of support for non-injective

transformations and the complexity of the specification of synchronization rules have been identified as main obstacles. Based on this analysis, we have presented different approaches to address these blocking factors.

Our investigations on bridging technical spaces (cf. Chap. 5) did analyze concrete bridges for highly relevant spaces. We can conclude that bridging technical spaces is an essential requirement in the context of RTE, which demands for a substantial amount of manual work. The lack of formal descriptions of the properties of such technical spaces accounts for this.

Despite the required manual work, we have also seen that some bridges (e.g., the one between context-free grammars and EMOF) were particularly useful to treat a variety of software artifacts and therefore are worthwhile creating. Moreover, the experiences gained during the evaluation of our work confirm that the strict separation of transferring data to a common technical space and the logical synchronization of redundant data eases the creation of RTE solutions.

To perform logical synchronization, we have presented an approach that explicitly supports non-injective transformations (PREP, cf. Chap. 6). Here, our evaluation has both confirmed that such transformations are required in practical MDSD processes and that change translation as it is implemented by PREP is feasible to solve this problem. We consider this as a main contribution of the thesis.

Also, PREP relies on the primitive operators of transformation languages. Thus, it is independent of concrete transformation specifications, which we consider an essential advantage over the specification of inverse transformations. Even bidirectional transformation languages that encode the specification of forward and backward rules in a single rule, require developers to consider both transformation directions.

To avoid redundant artifacts and the implied synchronization issues by construction, we have investigated the feasibility of role modeling for tool design and integration. Albeit there are open questions w.r.t. the general applicability of this approach, we can conclude that role models allow to decouple the domain abstractions required by tools from the actual data they operate on. Refactory—our role-based model refactoring tool—is a great example for this separation of tools and data.

Our evaluation has shown how one can flexibly adapt tools (e.g., refactorings) to different data structures (i.e., heterogeneous models). We think that the same idea is already implicitly employed in practical tool integration scenarios. However, we are also convinced that a conceptual basis, as provided by role modeling, can substantially contribute to ease tool integration. Therefore, we consider the continuation of this idea to be very promising.

The application of semantic web technologies for the specification of synchronization rules (cf. Chap. 7) did not meet our high expectations. Although we were able to employ OWL and SWRL for the synchronization of models, we experienced many difficulties due to the large gap between the respective technical spaces. In addition, we used EMOF models that did not include formal model semantics. Thus, we could not facilitate

existing semantics and were required to write complete synchronization specifications. We could gain minor benefits (e.g., automatic consistency checks for synchronization rules), but not use the semantic technologies to their full extent.

We can conclude this thesis with the understanding that the design of RTE systems is an inherently complex problem that requires a thorough understanding of the individual steps that must be taken to successfully avoid or synchronize redundant data. We think that our analysis of technical spaces and the bridges between them can provide guidance for designers of RTE systems. Also, we have shown the practical feasibility of backpropagation-based RTE and role-based tool integration. Both approaches are generic and can thus be used for RTE scenarios beyond the ones presented in this thesis. We are convinced that their essential ideas will affect the future design of RTE systems.

11.3 Future Work

This thesis raised further questions, which could not be addressed yet. We will give a summary of these according to the structure of the main chapters. First, we identify questions raised by Chap. 5. Second, prospective paths for backpropagation-based RTE systems are sketched. Third, potential continuations for applying the semantic technologies from Chap. 7 are presented, which is followed by a list of open issues for role-based tool integration. General items for future work can be found at the end of this section.

Automated Bridge Building between Technical Spaces The bridges between technical spaces that were built during the context of this thesis required a lot of manual work. Building tools to ease the mapping of concrete languages (e.g., EMFText or OWLizer), was performed entirely manual. This was mainly caused be the lack of a formal description of the involved technical spaces [164, 154]. If such descriptions were available, the creation of bridge building tools can probably be automated, at least to a certain degree. Here, both the description of the properties of technical spaces requires future work, as well as how to employ these descriptions w.r.t. the automation of bridge building.

Also, more generic methods to extract data from technical spaces are required. For example, in [286] a configurable model extractor is presented, which allows to obtain different models from a given technical space by exploiting its reflection capabilities. To ease building bridges further investigations in this regard are required. The initial results from [286] show that configurable bridges could allow to bridge spaces more easily.

Beside the creation of bridges, we have seen that criteria and requirements for evaluating the quality and completeness of bridges is essential, in particular in the context of RTE. Thus, further investigations in this regard are required.

Extending Applicability of PREP Our backpropagation-based RTE approach (PREP) provides support for non-injective transformations, which is often required for practical

MDSD processes. However, we have considered mainly compositional transformations in this context, in particular ones that were based on ISC. While we have also applied the approach successfully to template-based code generation in Sect. 9.2, the question is which other compositional transformations are particularly relevant in MDSD. For example, integrating PREP in ATLAS Model Weaver (AMW) [287], which explicitly supports model composition, is a candidate for future evaluation.

Beside compositional transformations, the application of PREP to non-compositional transformations is an open research question. For example, widespread transformation languages like QVT and ATL need to be examined w.r.t. their behavior. Both employ pattern matching, which poses special issues when translating changes. The question, whether the growing number of change propagation options still allows for practical applications needs to be addressed here. Also, more complex traceability mechanisms (e.g., hyperedges) may provide better means to capture the complex relations between elements. Moreover, change propagation rules that refer to concrete transformation rules instead of basic transformation operators may be required here.

Since PREP depends on the basic operators that are employed by transformation systems, formal descriptions of such operators as proposed in [288] could be analyzed w.r.t. their feasibility in PREP. This would allow to automate the decision whether PREP is feasible for a particular transformation system or not.

One of the main advantages of PREP is its support for non-injective transformations, which is not provided by bidirectional transformation languages. However, PREP is also applicable to injective transformations. Thus, the question how bidirectional transformation approaches and PREP compare in the presence of an invertible transformation requires more investigations. We have provided some initial points in this regard in Sect. 6.5.1, but these observations were not exhaustive. In particular the question, which is more easy to realize from a practical perspective requires further attention.

Integrating Semantic Modeling and Round-Trip Engineering In Chap. 7, we employed semantic web technologies for the purpose of model synchronization. We have seen that the specification of formal semantics for concepts can be used to determine relations between domain models. However, we have created these specifications manually, because they were not readily available. We derived ontologies from EMOF metamodels, which focus on the definition of abstract syntax. Thus, neither information about the semantics of individual modeling languages nor across multiple ones was reused. If such cross-language semantics were provided by modeling languages, the benefit to be gained could certainly be increased. Synchronization rules that were required to be written manually, could be derived.

Large-scale Tool Integration with Role Models In Chap. 8, we have presented how to anticipate future RTE scenarios using role modeling. We have evaluated our approach

during the implementation of the Refactory tool (cf. Sect. 9.4). The approach proved to be very successful in this example. However, to gain more insights about the implications of role-based tool design, further case studies are needed. A first candidate for such an evaluation is the Dresden OCL toolkit[1]. The toolkit already employs a so-called pivot model, which allows to adapt the OCL interpreter to arbitrary modeling languages and repositories. We believe, that this pivot model can be expressed in terms of a role model and therefore serve as another example for role-based tool integration.

An aspect of role-based tool integration that has been briefly discussed in Sect. 8.5.3, but which was not relevant in our evaluation scenario, is data migration. Whenever a tool integration scenario evolves, the role bindings and groundings can change. Thus, the physical distribution of features across the integrated role models changes. For example, attributes may be stored in different role types. If one wants to migrate existing data, the changes applied to the role binding need to be reflected. This is similar to metamodel evolution, where existing models must be changed, when metamodels evolve.

For the Refactory tool, this aspect was not relevant, because we adapted the transformation that implements the refactoring to the models that were subject to refactorings. The grounding of the role models that correspond to the metamodels of the target languages in this case, was not changed. The physical representation of the models did not change. For other RTE applications this is not necessarily true.

Another import point for future work is the analysis of role bindings w.r.t. their RTE properties. We have seen that such bindings specify how to derive role features from player features. In general, this can require complex computations. We believe that dedicated role binding languages can foster this analysis. In contrast to using general purpose languages to implement role bindings, dedicated languages could allow to determine bindings which are information preserving and bindings which are not. In the latter case, tool integrators can still decide whether this is acceptable or not.

Miscellaneous In addition to the open issues that were identified for specific approaches presented in this thesis, we have collected a number of general questions that need to be addressed in the context of RTE.

First, we have seen examples where the relations between models do not allow to apply certain changes. For example, some types of changes may not be representable in corresponding models or require additional changes to yield a globally consistent set of models. To account for this situation, one could determine the set of valid edit operations and design editors such that invalid edits cannot be performed. This could provide a simple approach to avoid inconsistencies.

Second, we did not consider scenarios where models are edited in a distributed environment. Concurrent modifications pose additional challenges to build RTE systems. For example, less fine-grained information about applied changes can be available in such

[1] `http://www.dresden-ocl.org`

scenarios. Also, edits can conflict if models are not reconciled instantly. The former problem can also be observed if certain kinds of editors (e.g., textual ones) are used, which do not provide information about individual changes, but rather sequences of models. For the case of textual editors, the question is whether textual edits (i.e., additions and deletions of individual characters) can help to compute precise edit sequences.

Third, we assumed that the relations between models are known and can thus be captured in some kind of synchronization specification. In practice, these relations may not be obvious at first sight. Rather, redundancies are unconsciously introduced. Thus, the detection of redundant parts of models needs to be addressed. To do so, clone detection algorithms can be employed. Such algorithms have already been successfully applied to detect code clones and can probably be transferred to modeling languages. Also, custom model matching rules as proposed in [289] could help to discover relations between models and metamodels.

Appendix

A1 Reuseware Reuse Extension for State Machines

```
componentmodel org.reuseware.example.statemachines.statemachines
implements org.reuseware.lib.systems.default
epackages <http://www.emftext.org/language/minsm>
rootclass minsm::StateMachine {
  // Core
  fragment role Default if $not name.startsWith('advice')$ {
    port type Rec {
      minsm::StateMachine.states is hook {
        port = $self.name$
      }
    }

    port type Config {
      minsm::State is anchor {
        port = $self.name$
      }
      minsm::State.successors is slot {
        port = $self.name$
      }
    }
  }
  // Aspect
  fragment role Default if $name.startsWith('advice')$ {
    port type Contrib {
      minsm::State is prototype if $not (name.startsWith('in') or name.startsWith('out'))$ {
        port = $self.owner.oclAsType(minsm::StateMachine).name$
      }
    }

    port type Config {
      minsm::State.successors is anchor if $name.startsWith('in')$ {
        port = $self.name$
      }
      minsm::State is slot if $name.startsWith('out')$ {
        mode = $'bind'$
        port = $self.name$
      }
    }
  }
}
```

Listing A1.1: Reuse extension specification for state machines.

A2 Ontologies for Petri Nets and Toy Train Models

```
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: : <http://www.emftext.org/language/petrinet#>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: petrinet: <http://www.emftext.org/language/petrinet#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Prefix: owl: <http://www.w3.org/2002/07/owl#>

Ontology: <http://www.emftext.org/language/petrinet>

Class: PetriNet    SubClassOf: owl:Thing
Class: Node        SubClassOf: owl:Thing
Class: Arc         SubClassOf: owl:Thing
Class: Transition  SubClassOf: Node
Class: Place       SubClassOf: Node
Class: Token       SubClassOf: owl:Thing

DataProperty: PetriNet_name     Domain: PetriNet  Range: xsd:string
ObjectProperty: PetriNet_nodes  Domain: PetriNet  Range: Node
ObjectProperty: PetriNet_arcs   Domain: PetriNet  Range: Arc

DataProperty: Node_name  Domain: Node  Range: xsd:string

DataProperty: Arc_name      Domain: Arc  Range: xsd:string
ObjectProperty: Arc_target  Domain: Arc  Range: Node
ObjectProperty: Arc_source  Domain: Arc  Range: Node

ObjectProperty: Place_marking  Domain: Place  Range: Token

DataProperty: Token_name  Domain: Token  Range: xsd:string

DisjointClasses: Token, PetriNet, Node, Arc
DisjointClasses: Place, Transition

Class: __c__PetriNet0_max_1_name
Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
EquivalentTo: PetriNet and not ( PetriNet_name max 1 xsd:string )

Class: __c__Node1_max_1_name
Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
EquivalentTo: Node and not ( Node_name max 1 xsd:string )

Class: __c__Arc2_max_1_name
Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
EquivalentTo: Arc and not ( Arc_name max 1 xsd:string )

Class: __c__Arc3_min_1_target
Annotations: rdfs:comment "The min cardinality of \'1\' for reference \'target\' is not satisfied."
EquivalentTo: Arc and not ( Arc_target min 1 Node )

Class: __c__Arc4_max_1_target
Annotations: rdfs:comment "The max cardinality of \'1\' for reference \'target\' is not satisfied."
EquivalentTo: Arc and not ( Arc_target max 1 Node )

Class: __c__Arc5_min_1_source
Annotations: rdfs:comment "The min cardinality of \'1\' for reference \'source\' is not satisfied."
```

```
56 EquivalentTo: Arc and not ( Arc_source min 1 Node )
57
58 Class: __c__Arc6_max_1_source
59 Annotations: rdfs:comment "The max cardinality of \'1\' for reference \'source\' is not satisfied."
60 EquivalentTo: Arc and not ( Arc_source max 1 Node )
61
62 Class: __c__Token7_max_1_name
63 Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
64 EquivalentTo: Token and not ( Token_name max 1 xsd:string )
```

Listing A2.1: OWL ontology for Petri net metamodel.

```
1  Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  Prefix: : <http://www.emftext.org/language/rails#>
3  Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
4  Prefix: rails: <http://www.emftext.org/language/rails#>
5  Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
6  Prefix: owl: <http://www.w3.org/2002/07/owl#>
7
8  Ontology: <http://www.emftext.org/language/rails>
9
10 Class: Component   SubClassOf: owl:Thing
11 Class: Train       SubClassOf: owl:Thing
12 Class: Project     SubClassOf: owl:Thing
13 Class: Track       SubClassOf: Component
14 Class: Switch      SubClassOf: Component
15 Class: Port        SubClassOf: owl:Thing
16 Class: Connection  SubClassOf: owl:Thing
17 Class: Out         SubClassOf: Port
18 Class: In          SubClassOf: Port
19
20 DataProperty: Component_name      Domain: Component  Range: xsd:string
21 ObjectProperty: Component_trains  Domain: Component  Range: Train
22 ObjectProperty: Component_ports   Domain: Component  Range: Port
23
24 DataProperty: Train_name  Domain: Train  Range: xsd:string
25
26 ObjectProperty: Project_components   Domain: Project  Range: Component
27 DataProperty: Project_name           Domain: Project  Range: xsd:string
28 ObjectProperty: Project_connections  Domain: Project  Range: Connection
29
30 DataProperty: Port_name  Domain: Port  Range: xsd:string
31
32 ObjectProperty: Connection_from  Domain: Connection  Range: Out
33 ObjectProperty: Connection_to    Domain: Connection  Range: In
34 DataProperty: Connection_name    Domain: Connection  Range: xsd:string
35
36 DisjointClasses: Connection, Project, Component, Train, Port
37 DisjointClasses: Switch, Track
38 DisjointClasses: In, Out
39
40 Class: __c__Component0_max_1_name
41 Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
42 EquivalentTo: Component and not ( Component_name max 1 xsd:string )
43
44 Class: __c__Train1_max_1_name
45 Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
```

```
46 EquivalentTo: Train and not ( Train_name max 1 xsd:string )
47
48 Class: __c__Project2_max_1_name
49 Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
50 EquivalentTo: Project and not ( Project_name max 1 xsd:string )
51
52 Class: __c__Port3_max_1_name
53 Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
54 EquivalentTo: Port and not ( Port_name max 1 xsd:string )
55
56 Class: __c__Connection4_max_1_name
57 Annotations: rdfs:comment "The max cardinality of \'1\' for attribute \'name\' is not satisfied."
58 EquivalentTo: Connection and not ( Connection_name max 1 xsd:string )
59
60 Class: __c__Connection5_min_1_from
61 Annotations: rdfs:comment "The min cardinality of \'1\' for reference \'from\' is not satisfied."
62 EquivalentTo: Connection and not ( Connection_from min 1 Out )
63
64 Class: __c__Connection6_max_1_from
65 Annotations: rdfs:comment "The max cardinality of \'1\' for reference \'from\' is not satisfied."
66 EquivalentTo: Connection and not ( Connection_from max 1 Out )
67
68 Class: __c__Connection7_min_1_to
69 Annotations: rdfs:comment "The min cardinality of \'1\' for reference \'to\' is not satisfied."
70 EquivalentTo: Connection and not ( Connection_to min 1 In )
71
72 Class: __c__Connection8_max_1_to
73 Annotations: rdfs:comment "The max cardinality of \'1\' for reference \'to\' is not satisfied."
74 EquivalentTo: Connection and not ( Connection_to max 1 In )
```

Listing A2.2: OWL ontology for toy train metamodel.

List of Figures

List of Tables

List of Listings

List of Abbreviations

AGG	Attributed Graph Grammar System.
AMMA	ATLAS Model Management Architecture.
AMW	ATLAS Model Weaver.
AOP	Aspect-Oriented Programming.
API	Application Programming Interface.
AST	Abstract Syntax Tree.
ATL	Atlas Transformation Language.
Beaver	Bidirectional Weaver.
BOTL	Bidirectional Object-oriented Transformation Language.
CDO	Connected Data Objects.
CM	Consistency Management.
CMR	Common Modeling Runtime.
CORBA	Common Object Request Broker Architecture.
CSP	Communicating Sequential Processes.
CWM	Common Warehouse Metamodel.
DL	Description Logics.
DOM	Document Object Model.
DPO	Double Push-Out.
DSLs	Domain-Specific Languages.
EBNF	Extended Backus Naur Form.
EMF	Eclipse Modeling Framework.
EMOF	Essential Meta Object Facility.
EOL	Epsilon Object Language.
ERD	Entity Relationship Diagram.

FSML	Framework-Specific Modeling Language.
GED	Graph Edit Distance.
GG	Graph Grammar.
GMM	Global Model Management.
GPL	General Purpose Language.
GReAT	Graph Rewriting and Transformation.
GRS	Graph Rewriting Systems.
GTXL	Graph Transformation Exchange Language.
HLRS	High-Level Replacement Systems.
HTML	Hypertext Markup Language.
IDL	Interface Definition Language.
ISC	Invasive Software Composition.
JMI	Java Metadata Interface.
JPA	Java Persistence API.
KM3	Kernel Meta Meta Model.
KMF	Kernel Modeling Framework.
MDA	Model-Driven Architecture.
MDSD	Model-Driven Software Development.
MISTRAL	Multiple IntenSion TRAnsformation Language.
MODELPLEX	MODELling solution for comPLEX software systems.
MOF	Meta Object Facility.
MOLA	MOdel transformation LAnguage.
ObjectTeams	Eclipse Object Teams Project.
OCL	Object Constraint Language.
OMG	Object Management Group.
OWA	Open World Assumption.
OWL	Web Ontology Language.

Pellet	Pellet: OWL 2 Reasoner for Java.
PREP	Propagate Replay Evaluate Pick.
QVT	Query View Transformation.
QVTO	Query View Transformation Operational.
QVTR	Query View Transformation Relational.
RDF	Resource Description Framework.
RoCoLa	Role Composition Language.
RTE	Round-Trip Engineering.
SAX	Simple API for XML.
SOS	Systems of Systems.
SPO	Single Push-Out.
SQL	Structured Query Language.
SWRL	Semantic Web Rule Language.
SysML	Systems Modeling Language.
TGG	Triple Graph Grammar.
TRS	Term Rewrite Systems.
UML	Unified Modeling Language.
UNA	Unique Name Assumption.
USMO	Unified Software Modeling Ontology.
VIATRA	VIsual Automated model TRAnsformations.
WSDL	Web Services Description Language.
XMI	XML Metadata Interchange.
XML	Extensible Markup Language.
XSD	XML Schema Definition.
XSLT	Extensible Stylesheet Language Transformation.

Bibliography

[1] Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development. John Wiley & Sons (2006)

[2] Kleppe, A.: Software Language Engineering. Pearson Education (2009)

[3] Staron, M.: Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. [293] 57–72

[4] France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering 2007, Minneapolis, Minnesota (2007) 37–54

[5] Spanoudakis, G., Zisman, A.: Inconsistency Management in Software Engineering: Survey and Open Research Issues. In: Handbook of Software Engineering and Knowledge Engineering, World Scientific (2001) 329–380

[6] The Object Management Group: Meta Object Facility (MOF) Core Specification, version 2.0. Technical report (2006)

[7] W3C: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. Technical report (2009)

[8] Gazzaniga, M.S.: Mind Matters. Houghton Miffin Co., Boston (1988)

[9] Ludewig, J.: Models in software engineering - an introduction. Software and Systems Modeling **2** (2003)

[10] Reenskaug, T., Wold, P., Lehne, O.: Working with Objects: The OOram Software Engineering Method. Manning Publications, Greenwich, CT (1996)

[11] Stachowiak, H.: Allgemeine Modelltheorie. Springer (1973)

[12] Seidewitz, E.: What Models Mean. IEEE Software **20** (2003) 26–32

[13] Aßmann, U., Zschaler, S., Wagner, G.: Ontologies, Meta-Models, and the Model-Driven Paradigm. In Calero, C., Ruiz, F., Piattini, M., eds.: Ontologies for Software Engineering and Technology. Springer (2006)

[14] Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific (1997)

[15] Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In Gorrieri, R., Wehrheim, H., eds.: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems. Volume 4037 of Lecture Notes in Computer Science., Bologna, Italy, Springer (2006) 171–185

[16] Bézivin, J., Lemesle, R.: Ontology-based Layered Semantics for Precise OA&D Modeling. In Bosch, J., Mitchell, S., eds.: Object-Oriented Technologys. Volume 1357 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1998) 287–292

[17] Hildenbrand, T., Gitzel, R.: A Taxonomy of Metamodel Hierarchies. http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2005/993/ (2005)

[18] Harel, D., Rumpe, B.: Modeling Languages: Syntax, Semantics, and All That Stuff – Part I: The Basic Stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel (2000)

[19] Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. [305] 159–168

[20] Warmer, J., Kleppe, A.: Object Constraint Language 2.0. Addison Wesley Professional (2004)

[21] The Object Management Group: Unified Modeling Language (UML), Infrastructure, Version 2.3. Technical report (2010)

[22] The Object Management Group: Unified Modeling Language (UML), Superstructure, Version 2.3. Technical report (2010)

[23] The Object Management Group: OMG Systems Modeling Language, Version 1.1. Technical report (2008)

[24] Chen, P.P.: The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems **1** (1976) 9–36

[25] Derntl, M., Figl, K., Botturi, L., Boot, E.: A Classification Framework for Educational Modeling Languages in Instructional Design. In: IEEE International Conference on Advanced Learning Technologies (ICALT 2006), Kerkrade, The Netherlands, IEEE Computer Society (2006)

[26] Zamli, K.Z., Lee, P.A.: A Taxonomy of Process Modeling Languages. In: AICCSA '01: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, Washington, DC, USA, IEEE Computer Society (2001) 435–447

[27] Sassone, V., Nielsen, M., Winskel, G.: A Classification of Models for Concurrency. In Best, E., ed.: CONCUR'93: Proceedings of the 4th International Conference on Concurrency Theory. Springer, Berlin, Heidelberg (1993) 62–96

[28] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework (2nd Edition). Pearson Education (2009)

[29] Frank, U.: The MEMO Meta Modelling Language (MML) and Language Architecture. ICB-Research Report 24, Institut für Informatik und Wirtschaftsinformatik (ICB), Universität Duisburg-Essen (2008)

[30] Kelly, S.: The Model Repository: More than just XML under version control? (2008) Keynote at the 8th OOPSLA Workshop on Domain-Specific Modeling, Nashville, TN.

[31] Hein, C., Ritter, T., Wagner, M.: Model-Driven Tool Integration with ModelBus. In: 1st International Workshop on Future Trends of Model-Driven Development (FTMDD 2009). (2009)

[32] Dirckze, R.: Java Metadata Interface (JMI) Specification, Version 1.0. Technical report, Sun Microsystems (2002)

[33] Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In Rensink, A., Warmer, J., eds.: Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings. Volume 4066 of Lecture Notes in Computer Science., Springer (2006) 128–142

[34] Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2004)

[35] Allilaire, F., Bézivin, J., Brunelière, H., Jouault, F.: Global Model Management In Eclipse GMT/AM3. In: Proceedings of the Eclipse Technology eXchange workshop (ETX) at ECOOP'06. (2006)

[36] Nijssen, G.M., Halpin, T.: Conceptual Schema and Relational Database Design. Prentice-Hall (Australia) (1989)

[37] Kristensen, B.B.: Object-Oriented Modelling with Roles, Springer (1995) 57–71

[38] Lee, J.S., Bae, D.H.: An enhanced role model for alleviating the role-binding anomaly. Software, Practice and Experience **32** (2002) 1317–1344

[39] Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. PhD thesis, University of Twente, Enschede, The Netherlands (2005)

[40] Riehle, D., Gross, T.R.: Role Model Based Framework Design and Integration. In: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98). (1998) 117–133

[41] Balzer, S., Gross, T.R., Eugster, P.: A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. [294] 323–346

[42] Coulondre, S., Libourel, T.: Towards a New Role Paradigm for Object-Oriented Modeling. In Bruel, J.M., Bellahsene, Z., eds.: OOIS Workshops. Volume 2426 of Lecture Notes in Computer Science., Springer (2002) 44–52

[43] Ungar, D., Smith, R.B.: Self: The Power of Simplicity. In Meyrowitz, N.K., ed.: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), October 4-8, 1987, Orlando, Florida, USA, Proceedings. SIGPLAN Notices 22(12). (1987) 227–242

[44] Herrmann, S.: Programming with Roles in ObjectTeams/Java (2005) AAAI Fall Symposium: Roles, An Interdisciplinary Perspective.

[45] Jørgensen, B.N.: Language Support for Incremental Integration of Independently Developed Components in Java. In: SAC '04: Proceedings of the 2004 ACM symposium on Applied computing, New York, NY, USA, ACM (2004) 1316–1322

[46] Baldoni, M., Boella, G., van der Torre, L.: Roles as a Coordination Construct: Introducing powerJava. Electronic Notes in Theoretical Computer Science **150** (2006) 9–29

[47] Tamai, T., Ubayashi, N., Ichiyama, R.: An Adaptive Object Model with Dynamic Role Binding. In Roman, G.C., Griswold, W.G., Nuseibeh, B., eds.: 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, ACM (2005) 166–175

[48] Schrefl, M., Thalhammer, T.: Using roles in Java. Software—Practice & Experience **34** (2004) 449–464

[49] Bäumer, D., Riehle, D., Siberski, W., Wulf, M.: The Role Object Pattern. In: Proceedings of the 4th Pattern Languages of Programming Conference (PLoP'97), Washington University Dept. of Computer Science, Technical Report (wucs-97-34). (1997)

[50] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of reusable Object-Oriented Software. Addison-Wesley Professional (1995)

[51] Fowler, M.: Dealing with Roles. http://martinfowler.com/apsupp/roles.pdf (1997)

[52] Mossé, F.G.: Modeling Roles - A Practical Series of Analysis Patterns. Journal of Object Technology **1** (2002) 27–37

[53] Chandra Sekharaiah, K., Janaki Ram, D.: Object Schizophrenia Problem in Object Role System Design. In Bellahsène, Z., Patel, D., Rolland, C., eds.: Object-Oriented Information Systems. Volume 2425 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2002) 1–8

[54] W3C: SWRL: A Semantic Web Rule Language Combining OWL and RuleML - W3C Member Submission. Technical report (2004)

[55] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)

[56] Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American **284** (2001) 34–43

[57] W3C: Resource Description Framework (RDF): Concepts and Abstract Syntax. Technical report (2004)

[58] W3C: OWL 2 Web Ontology Language: Profiles. Technical report (2009)

[59] Horridge, M., Drummond, N., Goodwin, J., Rector, A.L., Stevens, R., Wang, H.: The Manchester OWL Syntax. In Grau, B.C., Hitzler, P., Shankey, C., Wallace, E., eds.: Proceedings of the OWLED'06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA, November 10-11, 2006. Volume 216 of CEUR Workshop Proceedings., CEUR-WS.org (2006)

[60] W3C: OWL 2 Web Ontology Language: Direct Semantics. Technical report (2009)

[61] Horrocks, I., Kutz, O., Sattler, U.: The Even More Irresistible SROIQ. In Doherty, P., Mylopoulos, J., Welty, C.A., eds.: Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006, AAAI Press (2006) 57–67

[62] W3C: Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical report (2008)

[63] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A Practical OWL-DL Reasoner. Web Semantics: Science, Services and Agents on the World Wide Web **5** (2007) 51–53

[64] Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software **20** (2003) 42–45

[65] Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture. (2003)

[66] Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal **45** (2006) 621–645

[67] Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation. In Karsai, G., Taentzer, G., eds.: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). Volume 152 of Electronic Notes in Theoretical Computer Science., Elsevier (2006) 125–142

[68] Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. [299]

[69] The Object Management Group: Common Warehouse Metamodel (CWM) Specification, v1.1. Technical report (2003)

[70] W3C: XSL Transformations (XSLT) Version 1.0. Technical report (1999)

[71] Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer, Berlin (2003)

[72] Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury: an efficient purely declarative logic programming language. Journal of Logic Programming **29** (1996) 17–64

[73] Kifer, M., Lausen, G.: F-Logic: A Higher-Order language for Reasoning about Objects, Inheritance, and Scheme. In Clifford, J., Lindsay, B.G., Maier, D., eds.: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989, ACM (1989) 134–146

[74] Cordy, J.R.: The TXL Source Transformation Language. Science of Computer Programming **61** (2006) 190–210

[75] Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming **72** (2008) 52–70 Special Issue on Second issue of experimental software and toolkits (EST).

[76] Visser, E.: Meta-Programming with Concrete Object Syntax. In Batory, D.S., Consel, C., Taha, W., eds.: Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA,

October 6-8, 2002, Proceedings. Volume 2487 of Lecture Notes in Computer Science., Springer (2002) 299–315

[77] Blostein, D., Schürr, A.: Computing with Graphs and Graph Rewriting. Software - Practice and Experience **29** (1999) 1–21

[78] Baresi, L., Heckel, R.: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. [299] 402–429

[79] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings. Volume 903 of Lecture Notes in Computer Science., Springer Verlag (1994)

[80] Grunske, L., Geiger, L., Lawley, M.: A Graphical Specification of Model Transformations with Triple Graph Grammars. In Hartman, A., Kreische, D., eds.: Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings. Volume 3748 of Lecture Notes in Computer Science., Springer (2005)

[81] Königs, A.: Model Transformation with Triple Graph Grammars. In: Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica. (2005)

[82] Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electronic Notes in Theoretical Computer Science **148** (2006) 113–150

[83] The Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification Version 1.1. Technical report (2009)

[84] Greenyer, J., Kindler, E.: Reconciling TGGs with QVT. [307] 16–30

[85] Rensink, A., Nederpel, R.: Graph Transformation Semantics for a QVT Language. [303] 51–62

[86] Giandini, R.S., Pons, C., Pérez, G.: A two-level formal semantics for the QVT language. In Brogi, A., Araújo, J., Anaya, R., eds.: Memorias de la XII Conferencia Iberoamericana de Software Engineering (CIbSE 2009), Medellín, Colombia, Abril 13-17, 2009. (2009) 73–86

[87] Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. [307] 1–15

[88] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming **72** (2008) 31–39 Special Issue on Second issue of experimental software and toolkits (EST).

[89] Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. [301] 33–46

[90] Jouault, F., Kurtev, I.: On the Architectural Alignment of ATL and QVT. [292] 1188–1195

[91] Troya, J., Vallecillo, A.: Towards a Rewriting Logic Semantics for ATL. In Tratt, L., Gogolla, M., eds.: Proceedings of the Third international conference on Theory and practice of model transformations (ICMT'10). Volume 6142 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 230–244

[92] Jouault, F.: Loosely Coupled Traceability for ATL. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany (2005)

[93] Kurtev, I., van den Berg, K.: MISTRAL: A Language for Model Transformations in the MOF Meta-modeling Architecture. [301] 139–158

[94] Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. [301] 62–76

[95] Agrawal, A., Karsai, G., Shi, F.: Graph Transformations on Domain-Specific Models. Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University (2003)

[96] Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98, Paderborn, Germany, November 16-20, 1998, Selected Papers. Volume 1764 of Lecture Notes in Computer Science., Springer (1998) 296–309

[97] Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A.: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. International Journal on Software Tools for Technology Transfer (STTT) **6** (2004) 203–218

[98] Giese, H., Hildebrandt, S., Seibel, A.: Improved Flexibility and Scalability by Interpreting Story Diagrams. In Magaria, T., Padberg, J., Taentzer, G., eds.: Graph Transformation and Visual Modeling Techniques (GT-VMT 2009). Volume 18., Electronic Communications of the EASST (2009)

[99] Nickel, U.A., Niere, J., Wadsack, J.P., Zündorf, A.: Roundtrip Engineering with FUJABA (Extended Abstract). In: 2. Workshop Software-Reengineering, May 11-12, 2000, Bad Honnef, Germany. (2000)

[100] Priesterjahn, C., Tichy, M.: Modeling Safe Reconfiguration with the FUJABA Real-Time Tool Suite. [302]

[101] Ranger, U., Weinell, E.: The Graph Rewriting Language and Environment PROGRES. [297] 575–576

[102] Tichy, M., Meyer, M., , Giese, H.: On Semantic Issues in Story Diagrams. In Giese, H., Westfechtel, B., eds.: Proceedings of the 4th International Fujaba Days 2006, Bayreuth, Germany. (2006)

[103] Taentzer, G.: Towards Common Exchange Formats for Graphs and Graph Transformation Systems. Electronic Notes in Theoretical Computer Science **44** (2001) 28–40

[104] Lambers, L.: A New Version of GTXL : An Exchange Format for Graph Transformation Systems. Electronic Notes in Theoretical Computer Science **127** (2005) 51–63

[105] Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz, J.L., Nagl, M., Böhlen, B., eds.: Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers. Volume 3062 of Lecture Notes in Computer Science., Springer (2003) 446–453

[106] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)

[107] Tratt, L.: A change propagating model transformation language. Journal of Object Technology **7** (2008) 107–124

[108] Tratt, L.: The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London (2005)

[109] Schippers, H., Van Gorp, P., Janssens, D.: Leveraging UML Profiles to Generate Plugins From Visual Model Transformations. Electronic Notes in Theoretical Computer Science **127** (2005) 5–16

[110] Muliawan, O., Janssens, D.: Model refactoring using MoTMoT. International Journal on Software Tools for Technology Transfer **12** (2010) 201–209

[111] de Lara, J., Vangheluwe, H.: AToM^3: A Tool for Multi-formalism and Metamodelling. In Kutsche, R.D., Weber, H., eds.: Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings. Volume 2306 of Lecture Notes in Computer Science., Springer (2002) 174–188

[112] de Lara, J., Guerra, E.: Formal Support for Model Driven Development with Graph Transformation Techniques. In: Proceedings of Desarrollo Dirigido por Modelos. MDA y Aplicaciones. (DSDM'05), Granada, Spain. Volume 157., CEUR Workshop Proceedings (2005)

[113] Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In: 17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK, IEEE Computer Society (2002) 267–270

[114] The Object Management Group: Meta Object Facility (MOF) 2.0 XMI Mapping Specification, v2.1.1. Technical report (2007)

[115] Lawley, M., Steel, J.: Practical Declarative Model Transformation with Tefkat. [305] 139–150

[116] Aßmann, U.: Graph Rewrite Systems for Program Optimization. ACM Transactions on Programming Languages and Systems **22** (2000) 583–637

[117] Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency Checking of Conceptual Models via Model Merging. In: 15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India, IEEE Computer Society (2007) 221–230

[118] Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. [296] 42–51

[119] Easterbrook, S.: Model Management and Inconsistency in Software Design. In: NSF Workshop on the Science of Design. November 2-4, 2003, Airlie Center, Virginia, USA. (2003)

[120] Sendall, S., Küster, J.M.: Taming Model Round-Trip Engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development, Vancouver, Canada (2004)

[121] IEEE: IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990. Institute of Electrical & Electronics Engineers (1991)

[122] Letelier, P.: A Framework for Requirements Traceability in UML-based Projects. In: Proceedings of 1st International Workshop on Traceability in Emerging Forms of Software Engineering. In conjunction with the 17th IEEE International Conference on Automated Software Engineering, Edinburgh, U.K., September 2002. (2002) 32–41

[123] Ramesh, B., Stubbs, C., Powers, T., Edwards, M.: Requirements traceability: Theory and practice. Annuals of Software Engineering **3** (1997) 397–415

[124] Grammel, B., Kastenholz, S.: A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-driven Software Development. In Oldevik, J., Olsen, G.K., Kolovos, D.S., eds.: Proceedings of the 6th ECMFA Traceability Workshop (ECMFA-TW), 2010, Paris, France, June 15th, 2010. ACM International Conference Proceeding Series, New York, NY, USA, ACM (2010) 7–14

[125] Arnold, R., Bohner, S.: Software Change Impact Analysis. IEEE Computer Society (1996)

[126] Kassab, M., Ormandjieva, O., Daneva, M.: A Traceability Metamodel for Change Management of Non-functional Requirements. In Dosch, W., Lee, R.Y., Tuma, P., Coupaye, T., eds.: Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic, IEEE Computer Society (2008) 245–254

[127] Ramesh, B., Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering **27** (2001) 58–93

[128] Ibrahim, S., Idris, N.B., Munro, M., Deraman, A.: Integrating Software Traceability for Change Impact Analysis. International Arab Journal of Information Technology **2** (2005) 301–308

[129] Olsen, G.K., Oldevik, J.: Scenarios of Traceability in Model to Text Transformations. [291] 144–156

[130] Van Zuylen, H., ed.: The Redo Compendium: Reverse Engineering for Software Maintenance. John Wiley & Sons Ltd (1993)

[131] Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. Technical report, The Object Management Group (2003)

[132] Dijkstra, E.W.: On the Role of Scientific Thought. In: Selected Writings on Computing: A Personal Perspective. Springer-Verlag (1982) 60–66

[133] Tarr, P.L., Ossher, H., Harrison, W.H., Jr., S.M.S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In Boehm, B., Garlan, D., Kramer,

J., eds.: ICSE' 99. Proceedings of the 1999 International Conference on Software Engineering, May 16-22, 1999, Los Angeles, CA, USA, ACM (1999) 107–119

[134] de Lara, J., Guerra, E.: Generic Meta-modelling with Concepts, Templates and Mixin Layers. In Petriu, D.C., Rouquette, N., Haugen, Ø., eds.: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I. Volume 6394 of Lecture Notes in Computer Science., Springer (2010) 16–30

[135] Johannes, J., Aßmann, U.: Concern-Based (de)composition of Model-Driven Software Development Processes. [300] 47–62

[136] Aßmann, U.: Automatic Roundtrip Engineering. Electronic Notes in Theoretical Computer Science **82** (2003)

[137] Henriksson, A., Larsson, H.: A Definition of Round-Trip Engineering. Technical report, University of Linköping, Sweden (2003)

[138] Bancilhon, F., Spyratos, N.: Update Semantics of Relational Views. ACM Transactions on Database Systems **6** (1981) 557–575

[139] Dayal, U., Bernstein, P.A.: On the Correct Translation of Update Operations on Relational Views. ACM Transactions on Database Systems **7** (1982) 381–416

[140] Mayol, E., Teniente, E.: A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In Chen, P.P., Embley, D.W., Kouloumdjian, J., Liddle, S.W., Roddick, J.F., eds.: Advances in Conceptual Modeling: ER '99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling, Paris, France, November 15-18, 1999, Proceedings. Volume 1727 of Lecture Notes in Computer Science., Springer (1999) 62–73

[141] Hegner, S.J.: An Order-Based Theory of Updates for Closed Database Views. Annals of Mathematics and Artificial Intelligence **40** (2004) 63–125

[142] Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars. [306] 411–425

[143] Stevens, P.: Towards an Algebraic Theory of Bidirectional Transformations. [306] 1–17

[144] Demeyer, S., Ducasse, S., Tichelaar, S.: Why Unified is not Universal? UML Shortcomings for Coping with Round-trip Engineering. In France, R.B., Rumpe, B., eds.: UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings. Volume 1723 of Lecture Notes in Computer Science., Springer (1999) 630–644

[145] Angyal, L., Charaf, H.: Novel Techniques for Model-Code Synchronization. Electronic Communications of the EASST **8** (2008)

[146] Angyal, L., Lengyel, L., Charaf, H.: A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering. In: 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2008), 31 March - 4 April 2008, Belfast, Northern Ireland, IEEE Computer Society (2008) 463–472

[147] Chalabine, M., Kessler, C.: A Formal Framework for Automated Round-Trip Software Engineering in Static Aspect Weaving and Transformations. [295] 137–146

[148] Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful Lenses for String Data. In Necula, G.C., Wadler, P., eds.: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, January 7-12, 2008, San Francisco, California, USA, ACM (2008) 407–419

[149] Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with Round-Trip Engineering. In: Proceedings of ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS). Lecture Notes in Computer Science, Springer (2006)

[150] Euzenat, J., Shvaiko, P.: Ontology Matching. Springer, Heidelberg (2007)

[151] Seifert, M., Katscher, S.: Debugging Triple Graph Grammar-based Model Transformations. In Aßmann, U., Johannes, J., Zündorf, A., eds.: Proceedings of 6th International Fujaba Days, 18-19 Sep 2008, Dresden, Germany. (2008)

[152] Seifert, M., Werner, C.: Specification of Triple Graph Grammar Rules using Textual Concrete Syntax. [302]

[153] W3C: XML Schema Part 0: Primer Second Edition. Technical report (2004)

[154] Bézivin, J., Kurtev, I.: Model-based Technology Integration with the Technical Space Concept. In: Metainformatics Symposium, Springer-Verlag (2005)

[155] 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. In Palsberg, J., Abadi, M., eds.: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, ACM (2005) 233–246

[156] Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings. Volume 5063 of Lecture Notes in Computer Science., Springer-Verlag (2008) 31–45

[157] Priss, U.: Facet-like Structures in Computer Science. Axiomathes **18** (2008) 243–255

[158] Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting Object Interactions Using Composition Filters. In Nierstrasz, O., Guerraoui, R., Riveill, M., eds.: Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming. Volume 791 of Lecture Notes in Computer Science., Springer (1994) 152–184

[159] Smaragdakis, Y., Batory, D.: Mixin Layers - An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Transactions on Software Engineering and Methodology **11** (2002) 215–255

[160] Aßmann, U.: Invasive Software Composition. 1st edn. Springer Verlag (2003)

[161] McMenamin, S.P., Palmer, J.: Essential Systems Analysis. Yourdon Press (1984)

[162] Zhang, T., Jouault, F., Bézivin, J., Zhao, J.: A MDE Based Approach for Bridging Formal Models. In: Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, Washington, DC, USA, IEEE Computer Society (2008) 113–116

[163] Sun, Y., Demirezen, Z., Jouault, F., Tairas, R., Gray, J.: A Model Engineering Approach to Tool Interoperability. In Gasevic, D., Lämmel, R., Wyk, E.V., eds.: Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers. Volume 5452 of Lecture Notes in Computer Science., Springer (2009) 178–187

[164] Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. In: International Symposium on Distributed Objects and Applications, DOA Federated Conferences, Industrial track, Irvine, 2002. (2002)

[165] Bézivin, J., Devedzic, V., Djuric, D., Favreau, J.M., Gasevic, D., Jouault, F.: An M3-Neutral Infrastructure for Bridging Model Engineering and Ontology Engineering. In Konstantas, D., Bourrières, J.P., Léonard, M., Boudjlida, N., eds.: Interoperability of Enterprise Software and Applications. Springer London (2006) 159–171

[166] The Object Management Group: Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 1: CORBA Interfaces. Technical report (2008)

[167] The Object Management Group: Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 2: CORBA Interoperability. Technical report (2008)

[168] W3C: Web Services Description Language (WSDL) 1.1. Technical report (2001)

[169] Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., Piers, W.: Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In: Proceedings of the International Workshop on Software Factories at OOPSLA 2005. (2005)

[170] Brunelière, H., Cabot, J., Clasen, C., Jouault, F., Bézivin, J.: Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In Kühne, T., Selic, B., Gervais, M.P., Terrier, F., eds.: Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings. Volume 6138 of Lecture Notes in Computer Science., Springer (2010) 32–47

[171] W3C: Extensible Markup Language (XML) 1.0 (Initial Recommendation). Technical report (1998)

[172] W3C: Document Object Model (DOM) Level 1 Specification (Second Edition), Version 1.0. Technical report (2000)

[173] Brownell, D.: SAX. 1st edn. O'Reilly Media (2002)

[174] EJB 3.0 Expert Group: JSR 220: Enterprise JavaBeans, Version 3.0, Java Persistence API. Technical report (2006)

[175] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques, and Tools. Addison Wesley (1986)

[176] Wirth, N.: What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? Communications of the ACM **20** (1977) 822–823

[177] ISO/IEC: International Standard ISO/IEC 14977 - Syntactic metalanguage - Extended BNF. Technical report (1996)

[178] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In Paige, R.F., Hartman, A., Rensink, A., eds.: Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009.

Proceedings. Volume 5562 of Lecture Notes in Computer Science., Springer (2009) 114–129

[179] Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Construct to Reconstruct - Reverse Engineering Java Code with JaMoPP. [290]

[180] Calvanese, D., Lenzerini, M., Nardi, D.: Description Logics for Conceptual Data Modeling. In Chomicki, J., Saake, G., eds.: Logics for Databases and Information Systems. Kluwer Academic Publisher (1998) 229–263

[181] Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. [293] 528–542

[182] W3C: XML Schema Part 2: Datatypes Second Edition. Technical report (2004)

[183] Halpin, T.A.: A Logical Analysis of Information Systems: static aspects of the data-oriented perspective. PhD thesis, University of Queensland (1989)

[184] Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data Knowledge Engineering **35** (2000) 83–106

[185] Becker, S.M., Haase, T., Westfechtel, B.: Model-Based A-Posteriori Integration of Engineering Tools for Incremental Development Processes. Software and Systems Modeling **4** (2005) 123–140

[186] Antkiewicz, M., Czarnecki, K.: Design Space of Heterogeneous Synchronization. In Lämmel, R., Visser, J., Saraiva, J., eds.: Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007, Revised Papers. Volume 5235 of Lecture Notes in Computer Science. Springer, Berlin, Heidelberg (2008) 3–46

[187] Johannes, J., Samlaus, R., Seifert, M.: Round-trip Support for Invasive Software Composition Systems. In: International Conference on Software Composition 2009, SC 2009, 2 - 3 July 2009, Zurich, Switzerland. (2009)

[188] Henriksson, J., Johannes, J., Zschaler, S., Aßmann, U.: Reuseware – Adding Modularity to Your Language of Choice. Journal of Object Technology **6** (2007) 127–148 Special issue TOOLS Europe 2007.

[189] Tratt, L.: Model transformations and tool integration. Journal of Software and Systems Modelling **4** (2005) 112–122

[190] Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational Lenses: a Language for Updatable Views. In: PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, ACM (2006) 338–347

[191] Steimann, F.: UML-A oder warum die Wissenschaft ihre eigene einheitliche Modellierungssprache haben sollte. In Rumpe, B., Hesse, W., eds.: Modellierung. Volume 45 of Lecture Notes in Informatics., GI (2004) 121–133

[192] Johannes, J.: Component-Based Model-Driven Software Development. PhD thesis, TU Dresden (2010)

[193] Gao, X., Xiao, B., Tao, D., Li, X.: A survey of graph edit distance. Pattern Analysis & Applications **13** (2010) 113–129

[194] Alanen, M., Porres, I.: Difference and Union of Models. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings. Volume 2863 of Lecture Notes in Computer Science., Springer (2003) 2–17

[195] Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-Oriented Design Differencing. In Redmiles, D.F., Ellman, T., Zisman, A., eds.: 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA, ACM (2005) 54–65

[196] Lin, Y., Gray, J., Jouault, F.: DSMDiff: a differentiation tool for domain-specific models. European Journal of Information Systems **16** (2007) 349–361

[197] Abi-Antoun, M., Aldrich, J., Nahas, N.H., Schmerl, B.R., Garlan, D.: Differencing and Merging of Architectural Views. Automated Software Engineering **15** (2008) 35–74

[198] Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different Models for Model Matching: An analysis of approaches to support model differencing. In: International Workshop on Comparison and Versioning of Software Models, MCVS'09 at ICSE'09, IEEE Computer Society (2009)

[199] Rieckhof, F., Seifert, M., Aßmann, U.: Ontology-based Model Synchronisation. In Parreiras, F.S., Pan, J.Z., Aßmann, U., eds.: Third Workshop on Transforming and Weaving OWL Ontologies in MDE/MDA (TWOMDE 2010). (2010)

[200] Walter, T., Parreiras, F.S., Staab, S.: OntoDSL: An Ontology-Based Framework for Domain-Specific Languages. [298] 408–422

[201] Parreiras, F.S., Staab, S.: Using ontologies with UML class-based modeling: The TwoUse approach. Data & Knowledge Engineering **69** (2010) 1194–1207 Special issue on contribution of ontologies in designing advanced information systems.

[202] Lochmann, H.: HybridMDSD: Multi-Domain Engineering with Model-Driven Software Development using Ontological Foundations. PhD thesis, TU Dresden (2009)

[203] Bräuer, M., Lochmann, H.: An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning. In Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M., eds.: The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings. Volume 5021 of Lecture Notes in Computer Science., Springer (2008) 34–48

[204] Seifert, M., Wende, C., Aßmann, U.: Anticipating Unanticipated Tool Interoperability using Role Models. [296] 52–60

[205] Barbero, M., Jouault, F., Gray, J., Bézivin, J.: A Practical Approach to Model Extension. [291] 32–42

[206] 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.: Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings. Volume 3256 of Lecture Notes in Computer Science., Springer (2004) 287–303

[207] Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. [303] 159–170

[208] Andersen, E.P.: Conceptual Modeling of Objects: A Role Modeling Approach. PhD thesis, University of Oslo, Oslo, Norway (1997)

[209] Wende, C., Thieme, N., Zschaler, S.: A Role-based Approach Towards Modular Language Engineering. [304] 254–273

[210] Kiczales, G., Des Rivieres, J., Bobrow, D.: The Art of the Metaobject Protocol. The MIT Press (1991)

[211] Herrmann, S.: Object Teams: Improving Modularity for Crosscutting Collaborations. In Aksit, M., Mezini, M., Unland, R., eds.: Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers. Volume 2591 of Lecture Notes in Computer Science., Springer (2002) 248–264

[212] Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, University of Paderborn, D-33098 Paderborn, Germany (2007)

[213] Reimann, J., Seifert, M., Aßmann, U.: Role-Based Generic Model Refactoring. [300] 78–92

[214] Böhme, M.: Round-Trip Engineering für template-basierte Codegeneratoren. Diplomarbeit, TU Dresden (2009)

[215] Heidenreich, F., Johannes, J., Seifert, M., Wende, C., Böhme, M.: Generating Safe Template Languages. In Siek, J.G., Fischer, B., eds.: Generative Programming and Component Engineering, 8th International Conference, GPCE 2009, Denver, Colorado, USA, October 4-5, 2009, Proceedings, ACM (2009) 99–108

[216] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional (2000)

[217] Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. [304] 374–383

[218] Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: JaMoPP: The Java Model Parser and Printer. Technical Report TUD-FI09-10, Technische Universität Dresden, Fakultät Informatik (2009)

[219] Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall (1981)

[220] Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. [299] 161–176

[221] Lehman, M.M.: On understanding laws, evolution, and conservation in the large-program life cycle. Journal of Systems and Software **1** (1980) 213–221

[222] William F. Opdyke: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)

[223] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman, Amsterdam (1999)

[224] Taentzer, G., Müller, D., Mens, T.: Specifying Domain-Specific Refactorings for AndroMDA Based on Graph Transformation. [297] 104–119

[225] Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. [293] 425–439

[226] Naouel Moha, Vincent Mahé, Olivier Barais, Jean-Marc Jézéquel: Generic Model Refactorings. [298] 628–643

[227] Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems. Addison-Wesley (2007)

[228] Spanoudakis, G., Zisman, A.: Inconsistency Management in Software Engineering: Survey and Open Research Issues. In: Handbook of Software Engineering and Knowledge Engineering, World Scientific (2001) 329–380

[229] Hunter, A., Nuseibeh, B.: Managing Inconsistent Specifications: Reasoning, Analysis, and Action. ACM Transactions on Software Engineering and Methodology (TOSEM) **7** (1998) 335–367

[230] Robinson, W.N., Fickas, S.: Supporting Multi-Perspective Requirements Engineering. In: Proceedings of the 1st International Conference on Requirements Engineering (ICRE'94), IEEE Computer Society (1994) 206–215

[231] Kleppe, A., Warmer, J.: Do MDA Transformations Preserve Meaning? An investigation into preserving semantics. In Evans, A., Sammut, P., Willans, J.S., eds.: Metamodelling for MDA Workshop, First International Workshop, York, UK, Proceedings. (2003) 12–22

[232] Diskin, Z.: Algebraic Models for Bidirectional Model Synchronization. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings. Volume 5301 of Lecture Notes in Computer Science., Springer (2008) 21–36

[233] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. International Journal of Software Engineering and Knowledge Engineering **2** (1992) 31–58

[234] Easterbrook, S.: Handling Conflict Between Domain Descriptions With Computer-Supported Negotiation. Knowledge Acquisition **3** (1991) 255–289

[235] Nuseibeh, B., Easterbrook, S.M., Russo, A.: Making inconsistency respectable in software development. Journal of Systems and Software **58** (2001) 171–180

[236] Egyed, A.: Instant Consistency Checking for the UML. In Osterweil, L.J., Rombach, H.D., Soffa, M.L., eds.: 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006, ACM (2006) 381–390

[237] Egyed, A.: Fixing Inconsistencies in UML Design Models. [295] 292–301

[238] Egyed, A., Letier, E., Finkelstein, A.: Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy, IEEE Computer Society (2008) 99–108

[239] Nissen, H.W., Jeusfeld, M.A., Jarke, M., Zemanek, G.V., Huber, H.: Managing Multiple Requirements Perspectives with Metamodels. IEEE Software **13** (1996) 37 –48

[240] Engels, G., Küster, J.M., Heckel, R., Groenewegen, L.: A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In: Proceedings of the 8th European Software Engineering Conference (ESEC) held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE). (2001) 186–195

[241] Heckel, R., Küster, J., Taentzer, G.: Towards Automatic Translation of UML Models into Semantic Domains. In Kreowski, H.J., ed.: Proceedings of ETAPS 2002 Workshop on Application of Graph Transformation, Grenoble, France, April 2002. (2002)

[242] Küster, J.M.: Towards Inconsistency Handling of Object-Oriented Behavioral Models. In Heckel, R., ed.: Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT'2004). Volume 109 of Electronic Notes in Theoretical Computer Science., Elsevier (2004) 57–69

[243] Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM **21** (1978) 666–677

[244] Finkelstein, A.C., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency Handling in Multiperspective Specifications. IEEE Transactions on Software Engineering **20** (1994) 569 –578

[245] Spanoudakis, G., Finkelstein, A., Till, D.: Overlaps in Requirements Engineering. Automated Software Engineering **6** (1999) 171–198

[246] Fradet, P., Metayer, D.L., Perin, M.: Consistency Checking for Multiple View Software Architectures. In Nierstrasz, O., Lemoine, M., eds.: Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings. Volume 1687 of Lecture Notes in Computer Science., Springer (1999) 410–428

[247] Poon, W.L., Finkelstein, A.: Consistency Management for Multiple Perspective Software Development. In: Joint proceedings of the second international software

architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, New York, NY, USA, ACM (1996) 192–196

[248] Easterbrook, S., Nuseibeh, B.: Using ViewPoints for Inconsistency Management. BCS/IEE Software Engineering Journal **11** (1996) 31–43

[249] Sabetzadeh, M.: Management of Incomplete and Inconsistent Views. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), September 18-22, 2006, Tokyo, Japan, IEEE Computer Society (2006) 339–342

[250] Salay, R., Chechik, M., Easterbrook, S., Diskin, Z., McCormick, P., Nejati, S., Sabetzadeh, M., Viriyakattiyaporn, P.: An Eclipse-based Tool Framework for Software Model Management. In: Proceedings of the 2007 Workshop on Eclipse Technology eXchange (ETX'07) at 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM (2007) 55–59

[251] Mens, T., Straeten, R.V.D., D'Hondt, M.: Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. [293] 200–214

[252] Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. [293] 543–557

[253] Jakob, J., Königs, A., Schürr, A.: Non-materialized Model View Specification with Triple Graph Grammars. Volume 4178 of Lecture Notes in Computer Science., Springer (2006) 321–335

[254] Jakob, J., Schürr, A.: View Creation of Meta Models by Using Modified Triple Graph Grammars. [303] 175–184

[255] Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In Dwyer, M.B., Lopes, A., eds.: Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings. Volume 4422 of Lecture Notes in Computer Science., Springer (2007) 72–86

[256] Jahnke, J.H., Schäfer, W., Zündorf, A.: A Design Environment for Migrating Relational to Object Oriented Database Systems, IEEE Computer Society (1996) 163–170

[257] Becker, S., Haase, T., Westfechtel, B., Wilhelms, J.: Integration Tools Supporting Cooperative Development Processes in Chemical Engineering. In: Proceedings of

the 6th World Conference on Integrated Design & Process Technology (IDPT '02). (2002)

[258] 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)

[259] Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations. In Tratt, L., Gogolla, M., eds.: Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings. Volume 6142 of Lecture Notes in Computer Science., Springer (2010) 61–76

[260] Marschall, F., Braun, P.: Model Transformations for the MDA with BOTL. Technical report, Univeristy of Twente (2003)

[261] Braun, P., Marschall, F.: Transforming Object Oriented Models with BOTL. Volume 72 of Electronic Notes in Theoretical Computer Science., Elsevier (2003) 103–117

[262] Braun, P., Marschall, F.: BOTL - The Bidirectional Object Oriented Transformation Language. Technical Report TUM-I0307, Technische Universität München (2003)

[263] Sendall, S.: Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance? In: OOPSLA 03 Workshop Generative Techniques in the Context of MDA. (2003)

[264] Larsson, H., Burbeck, K.: Codex - An Automatic Model View Controller Engineering System. In: Proceedings of Workshop on Model Driven Architecture: Foundations and Applications, June 26-27, 2003, University of Twente, Enschede, The Netherlands. (2003) 37–48

[265] Burbeck, K., Larsson, H.: Automatic Model View Controller Engineering. Master's thesis, Linköping University (2002)

[266] Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards Automatic Model Synchronization from Model Transformations. In Stirewalt, R.E.K., Egyed, A., Fischer, B., eds.: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA, ACM (2007) 164–173

[267] Xiong, Y., Hu, Z., Takeichi, M., Zhao, H., Mei, H.: On-Site Synchronization of Software Artifacts. Technical Report METR 2008-21, Department of Mathematical Informatics, University of Tokyo (2008)

[268] Keller, A.M.: Updating Relational Databases Through Views. PhD thesis, Stanford University (1995)

[269] Lechtenbörger, J.: The Impact of the Constant Complement Approach Towards View Updating. In: Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA, ACM (2003) 49–55

[270] Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: algebraic foundations and the tile notation. In: 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM), Vancouver, BC, Canada, IEEE Computer Society (2009) 7–12

[271] Pierce, B.C.: Basic Category Theory for Computer Scientists. The MIT Press (1991)

[272] Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. [294] 600–624

[273] Herrmannsdörfer, M., Benz, S., Jürgens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In Drossopoulou, S., ed.: ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings. Volume 5653 of Lecture Notes in Computer Science., Springer (2009) 52–76

[274] Boger, M., Sturm, T., Fragemann, P.: Refactoring Browser for UML. In Aksit, M., Mezini, M., Unland, R., eds.: Objects, Components, Architectures, Services, and Applications for a Networked World. Volume 2591 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2003) 366–377

[275] Şavga, I.: A Refactoring-Based Approach to Support Binary Backward-Compatible Framework Upgrades. PhD thesis, TU Dresden (2010)

[276] Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting Transparent Model Update in Distributed CASE Tool Integration. [292] 1759–1766

[277] Jossic, A., Didonet Del Fabro, M., Lerat, J.P., Bézivin, J., Jouault, F.: Model Integration with Model Weaving: a Case Study in System Architecture. In: International Conference on Systems Engineering and Modeling (ICSEM '07). (2007) 79–84

[278] Jouault, F., Guéguen, T.: Integration by Model-driven Virtual Tool. In: Proceedings of the 2nd ECMDA Workshop on Model-Driven Tool and Process Integration, colocated with the 5th ECMDA 2009, Enschede, The Netherlands. (2009)

[279] Bézivin, J., Brunelière, H., Cabot, J., Doux, G., Jouault, F., Sottet, J.S.: Model Driven Tool Interoperability in Practice. In Hein, C., Wagner, M., eds.: 3rd Workshop on Model-Driven Tool and Process Integration, Co-located with ECMFA 2010, 16th June 2010, Paris, France. (2010)

[280] Broy, M., Feilkas, M., Herrmannsdörfer, M., Merenda, S., Ratiu, D.: Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. Proceedings of the IEEE **98** (2010) 526 –545

[281] Jouault, F., Bézivin, J., Barbero, M.: Towards an advanced model-driven engineering toolbox. Innovations in Systems and Software Engineering **5** (2009) 5–12

[282] Mahe, V., Jouault, F., Brunelière, H.: Megamodeling Software Platforms: Automated Discovery of Usable Cartography from Available Metadata. [290]

[283] Paesschen, E.V., Meuter, W.D., D'Hondt, M.: SelfSync: A Dynamic Round-Trip Engineering Environment. In Johnson, R.E., Gabriel, R.P., eds.: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA, ACM (2005) 146–147

[284] Ungar, D., Smith, R.B.: Self. In Ryder, B.G., Hailpern, B., eds.: Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), 9-10 June, 2007, San Diego, California, USA, ACM (2007) 1–50

[285] Bojić, D., Velašević, D.: URCA Approach to Scenario-based Round-trip Engineering. In: OOPSLA 2000 Workshop on Scenario-based Round-trip Engineering, Minneapolis, October 2000. (2000)

[286] Bézivin, J., Chevrel, R., Brunelière, H., Jossic, A., Piers, W., Jouault, F.: ModelExtractor: an Automatic Parametric Model Extractor. In: The international workshop on Object-Oriented Reengineering (WOOR) at the ECOOP 2006 Conference, Nantes, France. (2006)

[287] Didonet Del Fabro, M., Bézivin, J., Jouault, F., Breton, E., Guillaume, G.: AMW: A Generic Model Weaver. In: Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). (2005)

[288] Kurtev, I., Didonet Del Fabro, M.: A DSL for Definition of Model Composition Operators. In: Proceedings of the 2nd Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD, ECOOP 2006, Nantes, France. (2006)

[289] Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: A Domain Specific Language for Expressing Model Matching. In: Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09). (2009)

[290] Moonen, L., Systä, T., eds.: Proceedings of the International Workshop on Reverse Engineering Models from Software Artifacts, R.E.M. 2008, Satellite event of the 16th Working Conference on Reverse Engineering (WCRE), October 15, 2009, Lille, France. (2009)

[291] Akehurst, D.H., Vogel, R., Paige, R.F., eds.: Model Driven Architecture- Foundations and Applications, Third European Conference, ECMDA-FA 2007, Haifa, Israel, June 11-15, 2007, Proccedings. Volume 4530 of Lecture Notes in Computer Science., Springer (2007)

[292] Haddad, H.M., ed.: Proceedings of the 2006 ACM Symposium on Applied Computing, New York, NY, USA, ACM (2006)

[293] Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings. Volume 4199 of Lecture Notes in Computer Science., Springer (2006)

[294] Ernst, E., ed.: ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings. Volume 4609 of Lecture Notes in Computer Science., Springer (2007)

[295] 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, IEEE Computer Society (2007)

[296] Bézivin, J., Soley, R.M., Vallecillo, A., eds.: Proceedings of the First International Workshop on Model-Driven Interoperability (MDI 2010), New York, NY, USA, ACM (2010)

[297] Schürr, A., Nagl, M., Zündorf, A., eds.: Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers. Volume 5088 of Lecture Notes in Computer Science., Springer (2008)

[298] Schürr, A., Selic, B., eds.: Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings. Volume 5795 of Lecture Notes in Computer Science., Springer (2009)

[299] Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G., eds.: Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 07-12, 2002, Proceedings. Volume 2505 of Lecture Notes in Computer Science., Springer (2002)

[300] Petriu, D.C., Rouquette, N., Haugen, Ø., eds.: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II. Volume 6395 of Lecture Notes in Computer Science., Springer (2010)

[301] Aßmann, U., Aksit, M., Rensink, A., eds.: Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers. Volume 3599 of Lecture Notes in Computer Science., Springer (2005)

[302] van Gorp, P., ed.: Proceedings of 7th International Fujaba Days, 16-17 Nov 2009, Eindhoven, The Netherlands. (2009)

[303] Bruni, R., Varró, D., eds.: Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'06), Vienna, Austria, 1-2 April 2006. Electronic Notes in Theoretical Computer Science, Elsevier (2008)

[304] van den Brand, M., Gaševic, D., Gray, J., eds.: Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009 Revised Selected Papers. Volume 5969 of Lecture Notes in Computer Science., Springer (2010)

[305] Bruel, J.M., ed.: Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers. Volume 3844 of Lecture Notes in Computer Science., Springer (2006)

[306] Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds.: Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings. Volume 5214 of Lecture Notes in Computer Science., Springer (2008)

[307] Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings. Volume 4735 of Lecture Notes in Computer Science., Springer (2007)