On Integrating Triple Graph Grammars and OCL for Model-Driven Development

Duc-Hanh Dang

Bremen
2009

On Integrating Triple Graph Grammars and OCL for Model-Driven Development

by Duc-Hanh Dang

Dissertation
in partial fulfillment of the requirements for the degree of
Doctor of Engineering
- Dr. Ing. -

Submitted to Fachbereich 3
(Mathematics & Computer Science)
University of Bremen
September 2009

Date of Defense: 25 September 2009

Referee: Prof. Dr. Martin Gogolla
Prof. Dr. Hans-Jörg Kreowski

Abstract

Software systems become more and more complex. Despite significant advances in code-centric technologies such as advanced programming languages and integrated development environments (IDEs), developing complex software systems is still a laborious task. Model-centric software development has emerged as a promising paradigm, indicating a shift from "code-centric" to "model-centric" engineering. This paradigm puts forward a necessity as well as a challenge of a formal foundation for presenting precisely models and supporting automatic model manipulations.

This thesis focuses on a model-driven approach in which metamodeling and model transformation are seen as the core of the approach. On the one hand, metamodeling, which is a means for defining modeling languages, aims to precisely present models for specific purposes. On the other hand, model transformation is a robust approach for (1) transforming models from one language into another language, (2) tackling the challenge how to choose a right level of abstraction, and to relate levels of abstraction with each other, and (3) supporting software evolution and maintenance.

We propose an integration of two light-weight formal methods, the Object Constraint Language (OCL) and Triple Graph Grammars (TGGs), in order to form a core foundation for the model-driven approach. TGGs incorporating OCL allow us to explain relationships between models in precise way. On this foundation, we develop a textual language in order to describe transformations and to realize transformations. From such a declarative description of transformations, we propose employing OCL in order to derive operational scenarios for model transformation. This results in an OCL-based framework for model transformation. This framework offers means for transformation quality assurance such as verifying model transformations in an on-the-fly way, checking well-formedness of models, and maintaining model consistency. We explore case studies showing the practical applicability of our approach. Especially, TGGs incorporating OCL allow us to describe the operation semantics of use cases in a precise way. This approach on the one hand can be broaden in order to describe the operational semantics of modeling languages. On the other hand, it allows us to generate scenarios as test cases, to validate system behavior, and to check the conformance between use case models and design models. This supports basic constructions of an automatic and semi-automatic design.

Acknowledgments

I would like to express my gratitude to my supervisor Prof. Dr. Martin Gogolla. Under your supervision I became stronger in developing ideas as well as joining in the research community. I have learned a lot from you not only in research but in education and in life. I especially thank Prof. Dr. Hans-Jörg Kreowski for his willingness to co-supervise me. Cooperating with you is an important source of inspiration for me to carry out the thesis.

At this time the memory of my early days in Bremen appears in my mind clearly. I thank Arne Lindow, Fabian Büttner, Birgit Michaelis, and Hoa Nhu Tran for helping me to adapt to the new study environment. I also thank Mirco Kuhlmann, Fabian Büttner, Lars Hamann, Birgit Michaelis, Natalie Joulavskaia, and Arne Lindow for a very happy time at the Database Systems Group. Discussions and daily conversations in the Mensa with you helped me a lot to get over difficulties in my thesis work.

I owe many thanks to Mirco Kuhlmann, Lars Hamann, Ha Manh Tran, Wahju Widjajanto, Michael Lund, and Margaret Parks for their willingness to review the draft of my thesis and provide comments. I sincerely thank Dr. Karsten Sohr and the members in the ORKA probject, Michael Drouineaud and Tanveer Mustafa, for our exciting discussions. What I have learned from you is really useful for my thesis work. My thanks also go to Dr. Lars Grunske, Dr. Dániel Varró, and Ha Manh Tran for helping me by sharing their rich research experiences.

It is an honor for me to receive the acceptance of Prof. Dr. Jan Peleska and Prof. Dr. Rolf Drechsler to be on my Ph.D. committee. It affects your vacation plans. Thank you very much for the unconditional help.

I extend also thanks to MOET and DAAD as the two crucial sources for my thesis work. I also thank DAAD for the practical support during my study in German such as annual meetings and policies in administration procedures.

The study time in Bremen has made me realize more than ever how much my family means to me. The tremendous sacrifices of my parents and sister for my good education, and the love and the happiness from my wife and son have made me strong and able to complete the thesis. They are the semantics behind all that I do. I dedicate this dissertation to them.

Bremen, October 2009 Duc-Hanh Dang

Table of Contents

Chapter 1

Introduction

Advances in hardware technology have enabled the development of software-based systems that collaborate to provide essential services to society. The features of software in these systems are often distribution, adaptability in real time, and reliability. Despite significant advances in the code-centric technology such as advanced programming languages and integrated development environments, developing such complex software systems is still a laborious task.

Model-Driven Engineering (MDE) has emerged as a promising paradigm for developing complex software systems, indicating a paradigm shift from "code-centric" to "model-centric" engineering. This paradigm refers to models as the primary artifacts of development, which allow us to describe systems at multiple levels of abstraction. The focus of this approach is to narrow the wide conceptual gap between problem and software implementation domains. The gap is often mentioned as a significant factor behind the difficulty of the development. The MDE paradigm tackles the gap by a systematic transformation from problem-level abstractions to software implementations, from models to running systems.

The MDE paradigm puts forward a necessary and a challenge of a formal foundation for presenting precisely models and supporting automatic model manipulation. The first major challenge that researchers face when attempting to realize the MDE vision is the modeling language challenge. This challenge arises from how to provide support for creating and using problem-level abstractions in modeling languages as well as for rigorously analyzing models. Here, models are used to capture developer intent precisely, rather than expressing it informally. This is referred to as a language-based abstraction for raising the level of abstraction of models. In line with this concern, MDE also aims to move models closer to the implementation platform using platform-based abstractions such as classes, components, and services [GSCK04]. Another major challenge for realizing the MDE vision is the model manipulation challenge [FR07]. The challenge arises from problems such as (1) how to define, analyze, and use model transformations, (2) how to

maintain trace links among model elements, (3) how to maintain consistency among models as view points or model versions, and (4) how to use models during run-time.

Metamodeling and model transformation are currently the backbone of model-centric software engineering. They originate from the best known MDE initiative, Model-Driven Architecture (MDA), which is proposed by the Object Management Group (OMG). The OMG vision of MDE is founded in standards like UML, MOF, and OCL for modeling and metamodeling and Query/View/Transformation (QVT) [OMG07a] for model transformation. On the one hand, metamodels are used to define modeling languages and Domain-Specific Languages (DSLs). On the other hand, model transformations are employed and described using transformation languages like the QVT language.

Many approaches have been proposed for model transformation. Most of them focus on the standard QVT. ATL [JABK08] and Kermeta [MFJ05] are well-known systems developed according to this standard. They allow the developer to precisely present models using metamodels and OCL. Another promising approach for model transformation is the one based on graph transformation. VMTS [LLC08], Fujaba [Wag06], GME [KSLB03], and VIATRA [VP03] are among many tools realizing this approach. Recently, Triple Graph Grammars (TGGs), introduced in [Sch95], have been a promising approach for explaining relationships between models. Several tools such as MOFLON [AKRS06] and AToM3 [dLV02] implement TGGs for model transformation. The advantage of graph transformation based approaches is that they have a formal foundation, which allows automatically analyzing transformations. Model transformation in TGGs can be bidirectional whereas we only have uni-directional model transformation with QVT-like languages. However, the expressive power of graph transformation rules is often weaker than QVT-like approaches, since QVT-like approaches often include the declarative language OCL that allows us to express properties and to navigate in complex models.

Research Goals and Contributions

The goal of this thesis is to define a formal foundation for a model-driven approach, which is considered as the core for realizing the MDE vision. We claim that the integration of TGGs and OCL can support such a formal foundation. Figure 1.1 illustrates our research context. On the left side, models at different levels of abstraction and relationships between them (transfor-

mation, conformance, and trace) are depicted. On the right side, the relation between models of modeling languages, e.g., use case models and design models in this case, are represented.

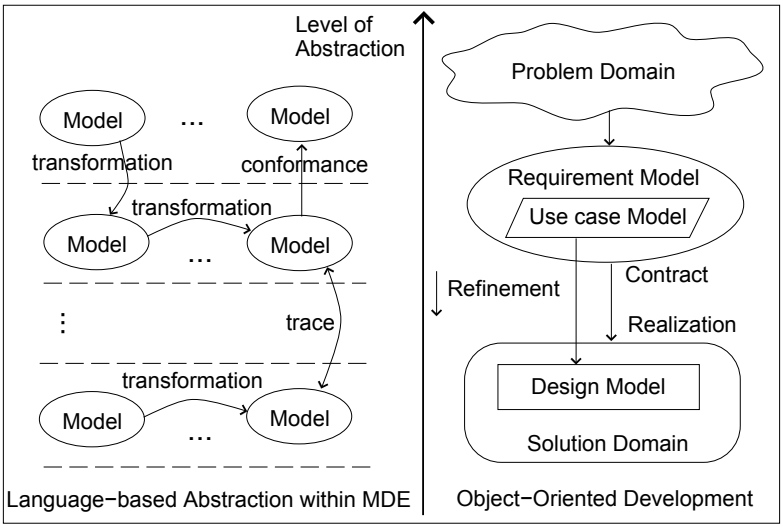


Figure 1.1: Relation between models in the MDE context in general and between use case models and design models in particular

Our work makes the following contributions:

- We define a formal foundation based on the incorporation of the object-oriented paradigm and triple graph transformation for a model-driven approach. For this foundation, we extend TGGs in several ways as follows: (1) We incorporate OCL in TGGs, (2) triple rules can be deleting rules together with multiple corresponding links, and (3) a new scenario derived from triple rules for model synchronization is proposed.

- On this foundation a USE-based tool for model transformation together with an OCL-based transformation assurance frame is developed. Specifically, we propose the USE4TGG language for realizing the integration of TGGs and OCL, resulting in the declarative description of transformations. We employ OCL in order to operationalize the integration of OCL and TGGs towards an OCL-based framework for model transformation. This OCL-based framework offers means for transformation quality assurance.

- We explore case studies showing the practical applicability of our approach. They include (1) the transformation between logic languages as introduced in Chapter 3, (2) the transformation from a UML subset

to a Java subset as explained in Chapter 5, (2) the transformation from UML activity diagrams to Communicating Sequential Processes (CSPs) as mentioned in Chapter 5 and in [DGB07, VAB+08], (4) the transformation between statecharts and extended hierarchical automatons as presented in [DG09b], and especially, (5) the relation between use case models and design models as explained in Chapter 6. The core of this case study is a method based on TGGs and OCL to relate scenarios at different levels of abstraction.

This thesis extends results that have in part been published. In particular the integration of TGGs and OCL is introduced in [DG09a, DG09b] and realized based on the work described in [DGB07, VAB+08, GBD08, MSD+08]. The key idea that we can use TGGs incorporating OCL to describe the semantics of use cases and to tackle the gap between use case and design models is described in [Dan07, Dan08, DG09c].

Thesis Structure

This thesis is organized as follows. The next two chapters, Chapters 2 and 3, describe a vision of model-driven development and explain a formal foundation for it based on the incorporation of the object-oriented paradigm and triple graph transformation. Chapter 4 describes the incorporation of OCL in TGGs. Chapter 5 explains how we can use OCL to operationalize TGGs incorporating OCL towards an OCL-based framework for model transformation. Chapter 6 focuses on the case study to define the operational semantics of use case. Finally, Chapter 7 summarizes the thesis and discusses directions for future research.

Chapter 2

Foundations for a Model-Driven Approach

This chapter focuses on a model-driven approach to complex engineering fields such as data engineering and software engineering. The core of this approach is that metamodels and models are used for model presentation, and transformations for model manipulation. A conceptual foundation for this approach can be defined as the incorporation of (1) the object-oriented paradigm (on which the Object Constraint Language (OCL) and the core of the Unified Modeling Language (UML) are founded) and (2) graph transformation.

2.1 Introduction

With the advent of graphic modeling languages like the Unified Modeling Language, models have been no longer considered mere as document elements, they can be explicitly manipulated by tools as full software artifacts. This is also the basic idea of Model-Driven Engineering (MDE), a promising methodology for software engineering. The best known MDE initiative is Model-Driven Architecture (MDA), the Object Management Group (OMG) vision of MDE. MDA is founded on standards like Meta-Object Facility (MOF), UML, and OCL for modeling and meta-modeling and Query/View/Transformation (QVT) for model transformations.

While the object-oriented paradigm can be seen as the formal foundation for the metamodeling and modeling techniques, to define a foundation for model transformation has been undergoing. Among different approaches to model transformation, graph transformation is a promising approach. This approach considers models in modeling languages as graphs.

This chapter aims to outline a conceptual foundation for the model-driven approach based on the incorporation of the object-oriented paradigm and graph transformation. This approach is not only applicable for software engineering but also for other engineering fields such as data engineering.

The structure of this chapter is as follows. Section 2.2 overviews the model-driven approach. Section 2.3 presents a formal foundation for the object-oriented paradigm. Section 2.4 and Sect. 2.5 briefly introduce the Unified-Modeling Language and the Object Constraint Language. Section 2.6 explains relevant works. This chapter is closed with a summary.

2.2 Overview of a Model-Driven Approach

This section explains fundamental concepts for a model-driven approach to complex engineering fields such as data engineering and software engineering.

2.2.1 Models and Metamodels

Many definitions of models, especially in the context of model-driven engineering have been introduced [Sei03, Rot89, KWB03, RJB04, Bé05]. We focus on a definition covering the OMG vision of models. The definition in [RJB04] is that "A *model* is a representation in a certain medium of something in the same or another medium. The model captures the important aspects of the thing being modeled from a certain point of view and simplifies or omits the rest." The medium for expressing models is convenient for working. It can be 3-D figures in a paper or a computer for models of buildings. It also can be modeling languages for data domains, languages, or software systems. The process resulting in models is called *modeling*.

It is often necessary to study the medium for expressing models before a modeling process. This activity often means a modeling of the medium. In this way we achieve a model of models, called a *metamodel*. The activity is called *metamodeling*. While a model is an abstraction of things in the real world, a metamodel is another abstraction, highlighting properties of the model itself. A model *conforms to* its metamodel only if properties of the model reflected in its metamodel are fulfilled. The relation between a model and its metamodel is close to the relation between a program and the programming language in which the program is written and defined by the grammar of the language.

The model-driven approach in our work employs the object-oriented paradigm in order to explain the conformance between a model and its metamodel. This approach covers the OMG vision of model-driven approaches. The original purpose of object-oriented paradigm is to address the complexity of a problem domain by considering the problem as a set of related, interact-

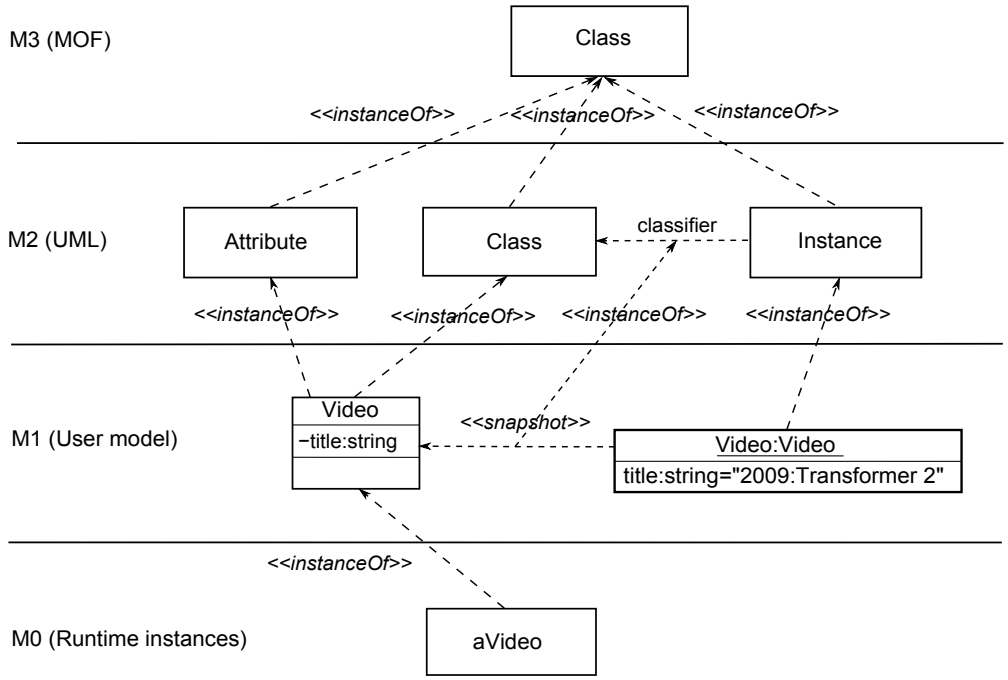


Figure 2.1: Example of the four-layer metamodel hierarchy of the UML [OMG07b, p.19]

ing *objects*. Each object represents an entity of interest in the system being modeled, and is characterised by its *class*, its state, and its behavior.

The use of this paradigm can be broaden for explaining the relationship between a model and its metamodel. Figure 2.1 shows an example of the four-layer metamodel hierarchy of the UML [OMG07b, p.19]. This infrastructure consists of a hierarchy of model levels, each (except the top) being characterized as "an instance" of the level above. As depicted in Fig. 2.1 level M0 holds the user data. Level M1 is a model of the data at level M0. Level M2 is a model of the model on level M1, therefore it is often referred to as a metamodel. Level M3 is said to hold a model of the information at level M2, and is often referred to as the meta-metamodel.

2.2.2 Modeling Languages and Metamodels

Modeling language is a means for expressing models. Modeling languages must be defined as rigorously as programming languages since models within model-driven approaches need to be processed by tools. In [KWB03] they are defined as follows: "A *modeling language* is a language with well-defined form (*syntax*), and meaning (*semantics*), which is suitable for automated

interpretation by a computer." A modeling language is often considered from two views: syntax and semantics. The syntax of a modeling language can be represented by graphical symbols or textual keywords. The language is called the graphical or textual modeling language, respectively. The semantics of a modeling language can be considered from two views [KM08]: The static semantics mentions well-formedness rules for models in the language, and the dynamic semantics mentions the meaning of the models.

The syntax of a modeling language may be abstract syntax or concrete syntax. The abstract syntax of a language describes the kinds of its elements in an abstract form, and the rules to combine those elements. In English, e.g., the kinds of elements include nouns, verbs, and adjectives, and the rules for combining them are given by English grammar. The concrete syntax of a language defines how the language elements appear in a concrete, human-usable form. A language may have many concrete syntax, similarly to English with two forms, the written (textual) and spoken forms.

There are different ways for defining the syntax of a modeling language. We mention two main approaches here, context-free grammars and metamodels. Context-free grammars (CFGs) are traditionally used to define the syntax (both concrete and abstract) of programming languages. The original purpose of CFGs is to process text, but they can also be used in the definition of graphical languages. A common notation for CFGs is the Backus-Naur form (BNF) notation. We will not go into further detail on the approach with CFGs and BNF since that is a widely understood subject.

Metamodeling is another approach to the definition of the syntax (often abstract syntax) of languages. The approach becomes popular since it was used to define the abstract syntax of UML. Metamodeling allows us to construct an object-oriented model of the abstract syntax of a language. For UML, this is expressed in a subset of UML. A metamodel presents language elements as classes and characterizes relationships between them using attributes and associations.

2.2.3 Model Transformation

Models play a fundamental role within model-driven approaches, and model transformation becomes a crucial activity for model manipulation. In software engineering essential aspects of software are expressed in the form of models, and transformations of these models are considered the core of software development. Models and transformations are used to specify, simulate,

test, and generate code.

In data engineering data models are the central subject. Data manipulations such as data translation, mapping, and integration are often complex since data come from various sources; They are often distributed and structured according to different data models such as ER, Relational Model, Object-Relational Model, XML, Excel sheets, Latex documents, Word documents, etc. When data models are studied by the metamodeling approach, model transformation can be used for such data manipulations. The OMG has proposed the standard Common Warehouse Metamodel (CWM) [OMG03a] in order to support for data transformation.

In language engineering modeling languages can be defined by metamodels as explained in Subsect. 2.2.2. Therefore, we can also use model transformation in order to translate between languages.

The OMG has proposed Query/Views/Transformation (QVT) as the standard language for specifying model transformations [OMG07a]. Many works have attempted to complement and realize QVT. Besides, there are many different approaches for model transformation [CH03]. Graph transformation for model transformation is the approach that can be compared to QVT. This approach is also the focus of our work.

The core of this approach is that within model-driven approaches based on the object-oriented paradigm, models and metamodels can be represented as UML object and class diagrams, respectively. Then, models can be considered as graphs. Model manipulations can be realized by graph transformation rules. The advantage of the approach is that graph transformation rules are easily and intuitively specified. Graph transformation supports a formal foundation for the approach. The complexity of the graph transformation rules as well as the application formalisms can be hidden from users.

2.3 Object-Oriented Paradigm

This section presents a formal foundation for the object-oriented paradigm. Such a formalization is proposed in [Ric02, RG98, OMG06] in order to define a formal semantics for the UML and OCL. We focus on this formalization as a foundation for presenting models within model-driven approaches.

2.3.1 Object Model

An object model represents information about structural aspects of a system. It is defined as a structure together with a type system such that the system at a particular moment in time can be seen as an illustration of the structure. Here, the system is viewed as a set of objects, and each object has attributes, behaviors, and relationships to other objects. For a representation of such a structure, an object model includes (1) a set of classes, (2) a set of attributes for each class, (3) a set of operations for each class, (4) a set of associations with role names and multiplicities, and (5) a generalization hierarchy over classes.

For an object model we assume that there is a signature $\Sigma = (T, \Omega)$ with T being a set of types, and Ω being a set of operations over types in T. The set T includes object types, types of attributes and operation parameters, and other types. In [Ric02, RG98, OMG06] the set T includes types in the type system of OCL, and then the object model becomes a context for the expression of OCL (of objects and relationships between objects). In this way OCL can be compared to first-order predicate logic about the expressiveness.

Definition 2.1. (Syntax of Object Models)
The syntax of an object model is a structure

$$\mathcal{M} = (\text{CLASS}, \text{ATT}_c, \text{OP}_c, \text{ASSOC}, \text{associates}, \text{roles}, \text{multiplicities}, \prec)$$

where

1. CLASS $\subseteq \mathcal{N}$ is a set of names representing a set of classes, where $\mathcal{N} \subseteq \mathcal{A}^+$ is a non-empty set of names over alphabet $\mathcal{A}$. Each class $c \in$ CLASS induces an object type $t_c \in T$. Values of an object type refer to objects of the class.

2. ATT_c is the attributes of a class $c \in$ CLASS, defined as a set of signatures $a : t_c \rightarrow t$, where the attribute name a is an element of $\mathcal{N}$, $t_c \in T$ is the type of class c, and $t \in T$ is the type of the attribute.

3. OP_c is a set of signatures for user-defined operations of a class c with type $t_c \in T$. The signatures are of the form $\omega : t_c \times t_1 \times ... \times t_n \rightarrow t$, where ω is the name of the operation, and t, t_1, ... , t_n are types in T.

4. ASSOC is a set of association names.

(a) associates : ASSOC $\rightarrow$ CLASS$^+$ is a function mapping each association name to a list of participating classes. This list has at least two elements.

(b) roles : ASSOC $\rightarrow \mathcal{N}^+$ is a function mapping each association to a list of role names. It assigns each class participating in an association a unique role name.

(c) multiplicities : ASSOC $\rightarrow \mathcal{P}(\mathbb{N}_0)^+$ is a function mapping each association to a list of multiplicities. It assigns each class participating in an association a multiplicity. A multiplicity is a non-empty set of natural numbers (an element of the power set $\mathcal{P}(\mathbb{N}_0)^+$) different from $\{0\}$.

5. $\prec$ is a partial order on CLASS reflecting the generalization hierarchy of classes. $\square$

Example. Figure 2.2 visualizes an object model in the form of a UML class diagram. A detailed explanation of this class diagram is shown in Sect. 2.4.

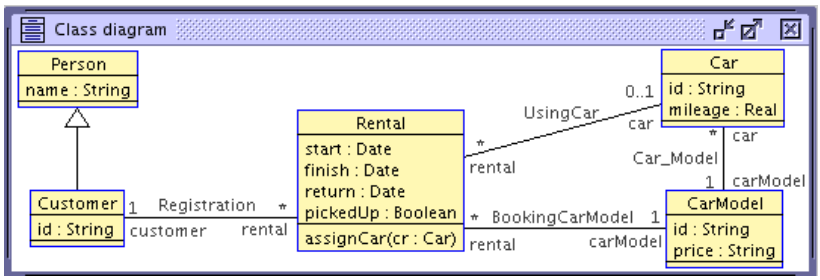


Figure 2.2: Object model visualized by a class diagram

2.3.2 Interpretation of Object Model

An interpretation of an object model is referred to as the state of the corresponding system at a particular moment in time. A system state is constituted by objects, links, and attribute values.

Definition 2.2. (System State)
A system state for a model $\mathcal{M}$ is a structure $\sigma(\mathcal{M}) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}})$ defined as follows.

1. For each $c \in$ CLASS, the finite set $\sigma_{\text{CLASS}}(c)$ contains all objects of class $c \in$ CLASS existing in the system state: $\sigma_{\text{CLASS}}(c) \subset oid(c)$.

2. Functions σ_{ATT} assign attribute values for each object in the state. $\sigma_{\text{ATT}}(a) : \text{CLASS}(c) \rightarrow I(t)$ for each $a : t_c \rightarrow \text{ATT}_c^*$.

3. For each $as \in \text{ASSOC}$, there is a set of current links: $\sigma_{\text{ASSOC}}(as) \subset I_{\text{ASSOC}}(as)$. A link set must satisfy all multiplicity specifications: $\forall i \in \{1, ..., n\}, \forall l \in \sigma_{\text{ASSOC}}(as)$: $|\{l' | l' \in \sigma_{\text{ASSOC}}(as) \wedge (\pi_i(l') = \pi_i(l))\}| \in \pi_i(\text{multiplicities}(as))$

where

- $I(t)$ is the domain of each type $t \in T$.
- $oid(c)$ is the infinite set of objects of each class $c \in \text{CLASS}$. $I_{\text{CLASS}}(c) = oid(c) \cup \{oid(c') | c' \in \text{CLASS} \wedge c' \prec c\}$.
- ATT_c^* is the set of direct and inherited attributes of class c: $\text{ATT}_c^* = \text{ATT}_c \cup_{c \prec c'} \text{ATT}_{c'}$.
- $I_{\text{ASSOC}}(as) = I_{\text{CLASS}}(c_1) \times ... \times I_{\text{CLASS}}(c_n)$ is the interpretation for each association $\text{associations}(as) = \langle c_1, c_2, ..., c_n \rangle$, where $as \in ASSOC$, and $c_1, c_2, ..., c_n$ are classes participating in the association. Each $l_{as} \in I_{\text{ASSOC}}(as)$ is referred to as a link.
- $\pi_i(l)$ projects the ith component of a list l. □

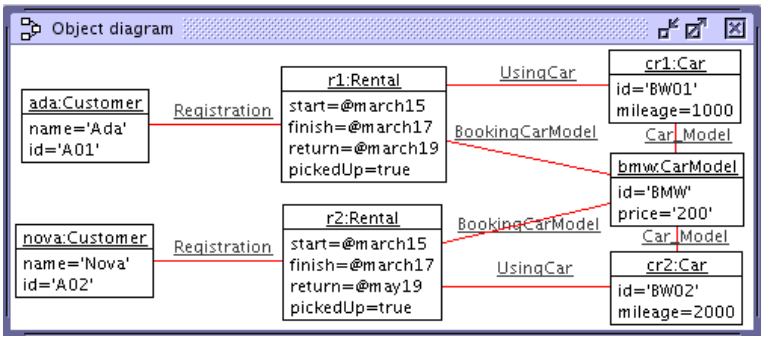


Figure 2.3: System state visualized by an object diagram

Example. Figure 2.3 visualizes a system state in the form of a UML object diagram. This system state is the interpretation of the object model shown in Fig. 2.2. A detailed explanation of this object diagram is shown in Sect. 2.4.

2.4 The Unified Modeling Language (UML)

This section briefly introduces the UML as a unified modeling language and as a means for defining metamodels. The available diagrams in the UML include use case diagrams, class diagrams, object diagrams, interaction diagrams, statecharts, activity diagrams, and deployment diagrams. Among these diagrams, we focus on the following diagrams.

- Class diagrams and object diagrams: They are basic constructs for defining system models and metamodels within model-driven approaches.
- Use case diagrams, interaction diagrams, and activity diagrams: They are the focus of our case study in Chapter 6. Use case diagrams will be explained in Chapter 6. The two remaining diagrams are introduced in the rest of this section.

For a full explanation of the available UML diagrams, we refer to (1) the infrastructure specification [OMG07b] which defines the foundation language constructs required for UML, and (2) the superstructure specification [OMG07c] which defines the user level constructs.

Class Diagram. A class diagram is a graphic presentation of the static structure view of the system model that shows the system's classes, their attributes, and the relationships between the classes. A *class* presents a concept within the system. It reflects a set of *objects* that share the same *attributes*, *operations*, and *relationships*. While an attribute describes values that objects may hold, an operation specifies the result of the behavior of objects.

Figure 2.2 shows an example for class diagrams. This diagram depicts several classes denoted by rectangles. The class name, the attributes, and the operations of each class are represented by compartments within each rectangle. For example, the `Rental` class has four attributes (described by the name and type of attributes) and the `assignCar(cr:Car)` operation.

Association and *generalization* are the most important relationships between classes. A generalization between two classes (the so-called superclass and subclass) means that objects of the subclass have the properties of the superclass and additional properties specific to the subclass. A generalization relationship is presented by a line from the subclass, e.g., the `Customer` class shown in Fig. 2.2, to the superclass, the `Person` class in this case, with a large hollow triangle at the end of the part.

An association is a relationship among classes, describing connections among their objects. Objects participating in an association can refer to each other through association ends together with *role names*. The *multiplicity* attached to an association end declares how many objects may fill the position defined by the association end. Figure 2.2 shows the `Registration` association between the `Customer` and `Rental` classes. This association indicates a `Rental` object can connect to one `Customer` object. A `Customer` object can connect to many `Rental` objects.

Object Diagram. The diagram is a graphic presentation representing the state of a running system at a specific time. This representation includes objects and links as instances of classes and associations specified in a class diagram. Objects are defined by concrete attribute values. A link connects the objects participating in the association.

Figure 2.3 shows an example for object diagrams. This object diagram conforms to the class diagram presented in Fig. 2.2.

Interaction Diagram. The diagram is used to specify how messages are exchanged over time between objects for a task. They include sequence diagram, communication diagram, and interaction overview diagram.

A sequence diagram shows an interaction arranged visually in time order. The objects participating in the interaction are shown by their lifelines and the messages they exchange. Each lifeline represents a role within the interaction. The diagram may includes fragments, which are the nested node of the tree that constructs interactions using control constructs such as loop, conditional, and parallel. The leaves of the interaction tree are *occurence specifications*, *execution specification*, and *constraints* on the state of the target instance and its parts. Figure 2.4 presents an example for sequence diagrams.

As another presentation of sequence diagrams, a communication shows an interaction arranged around the objects that perform operations. Unlike a sequence diagram, a communication diagram explicitly shows links among the communicating objects. The time sequence of messages in a communication diagram is determined using sequence numbers, whereas the message sequence within a sequence diagram is defined using the geometric order of the arrows in the diagram. Figure 2.5 shows an example for communication diagrams.

An interaction overview diagram is a variation on an activity diagram in which sequence diagram fragments are integrated into flow of control constructs.

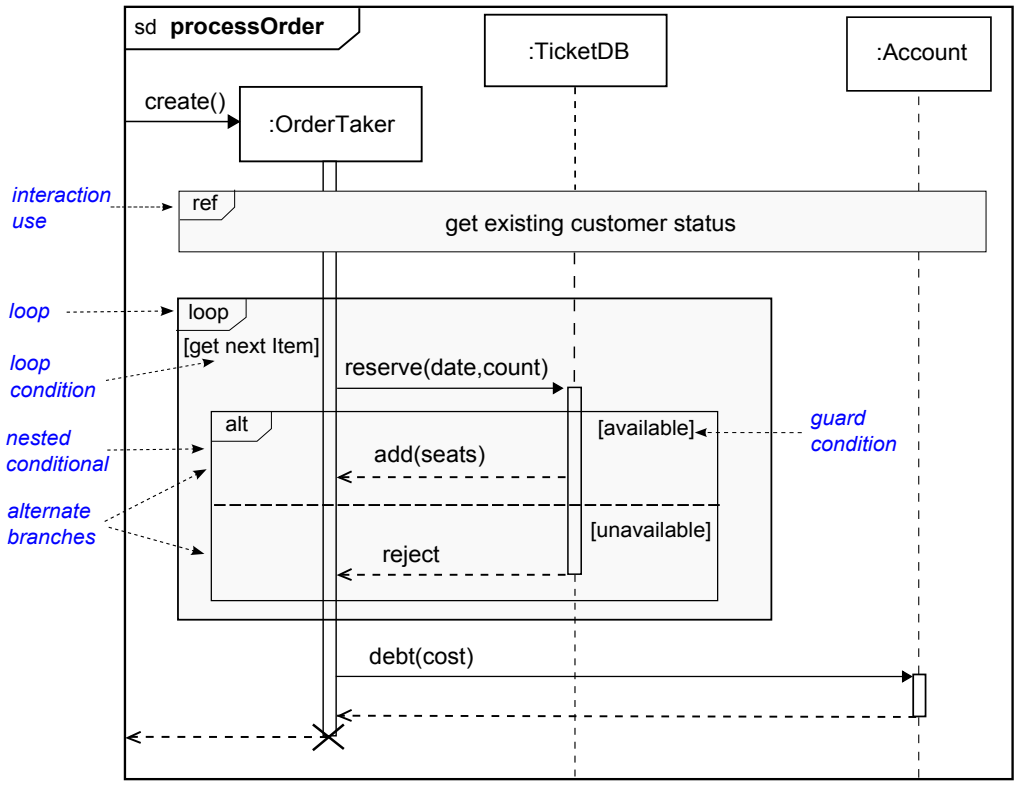


Figure 2.4: Example for sequence diagrams [RJB04, p.105]

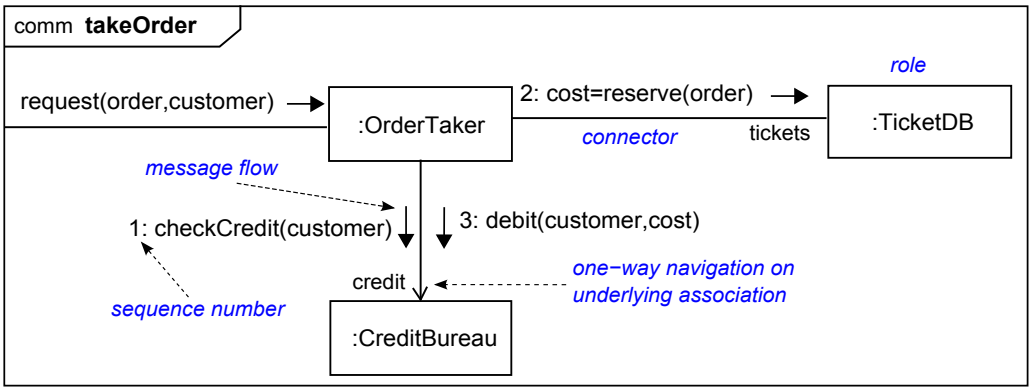


Figure 2.5: Example for communication diagrams [RJB04, p.107]

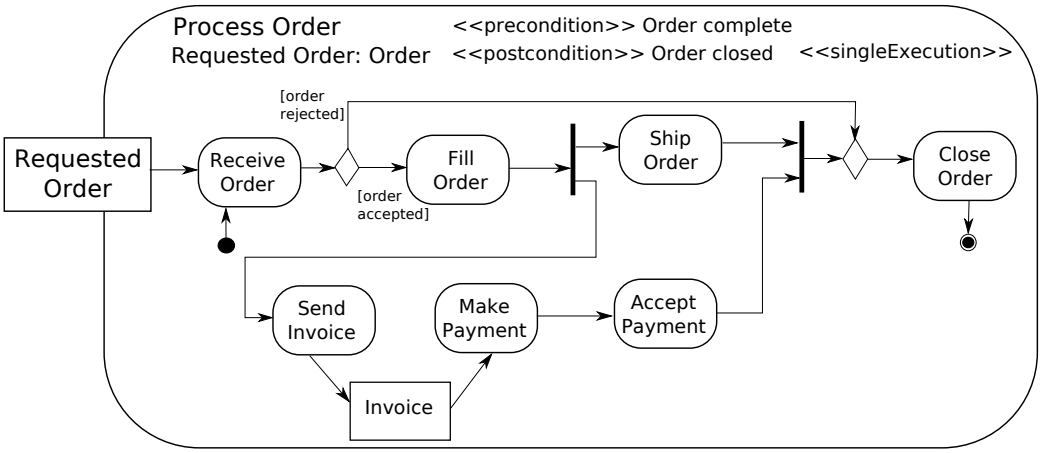


Figure 2.6: Example for activity diagrams [OMG07c, p.322]

Activity Diagram. The diagram is a graphic presentation of the decomposition of an activity into its constituents. Activity node and activity edge are two basic elements of an activity diagram. Activity nodes are connected by flows, which are represented by activity edges. Activity nodes can be actions, control nodes, object nodes, and structured nodes. The activity nodes within an activity diagram are often organized into partitions, often called swimlanes.

Figure 2.6 depicts an example for activity diagrams. Note that explicitly modeled actions as part of activities are new in UML 2.0. They represent a merger of activity graphs from UML 1.5 and actions from UML 1.5.

2.5 The Object Constraint Language

This section briefly introduces the Object Constraint Language (OCL). For a detailed explanation of the OCL, we refer to the work in [OMG06, Ric02, WK98].

2.5.1 Description of the OCL

The OCL is a formal language used to describe expressions on UML models. OCL expressions specify constraints on objects described in a model or queries over them. They do not have side effects, i.e., their evaluation cannot alter the corresponding system state.

The OCL is a typed language. Each valid (well-formed) OCL expression has a type, which is the type of the evaluated value of this expression. Two basic constructs in OCL are OCL expressions and OCL conditions. An OCL condition is referred to as an OCL expression of the Boolean type.

The type system of OCL can be divided into predefined types or types defined by the UML models. They include basic types (e.g., `Integer, Real, String, and Boolean`), object types, collection types (e.g., `Collection(t), Set(t), Bag(t), and Sequence(t)` for describing collections of values of type `t`), and message types. These types have operations over them. OCL operations are close to the ones in query languages such as SQL [GR97].

OCL expressions are often defined in the context of an instance of a specific type. We can access the contextual instance using the `self` name. OCL can be used for the following purposes.

- To specify invariants on classes and types in the class model. An invariant on a class is an OCL condition that must be true for all instances of this class in all system states. Example:

```
-- The number of cars in a car model is
-- higher than 10.
context CarModel inv:
self.car.size() > 10
```

- To describe pre- and post conditions on operations. They are OCL conditions that must be true before and respectively after the execution of the given operation. Example:

```
-- When a car is picked up.
context Rental::assignCar(cr:Car)
pre assignCarPre:  self.car = OclUndefined(Car)
post assignCarPost:  self.car = cr
```

- To describe guards. A guard is an OCL condition that determines whether or not the transaction in a state transition diagram is enabled.

- As a query language, i.e., to query over the given system state by OCL expressions.

Since UML class models can be used to define metamodels, e.g., MOF metamodels, we can use OCL to restrict on metamodels in order present precisely models. These OCL conditions are referred to as well-formedness rules.

2.5.2 OCL and Object Model

The OCL is developed on the basis of an object model. The aim is to allow us to express attribute values and logic conditions on the structure defined by the object model. Specifically, the object model structure can be extended with an OCL algebra as presented in the work [Ric02, RG98, OMG06]. In this extension types in the type system of OCL coincide with types (of the set T) in the object model.

Such a formalization of OCL mentioned above allows us to conclude in Chapter 3 that OCL expressions are also applicable to attributed graphs that conform to a corresponding object model.

2.6 Relevant Work

This section overviews relevant works for this chapter. For a definition of models, the work in [Rot89] focuses on explaining of the nature of modeling. The work in [Sei03] studies the definition of models and metamodels in relation to the OMG metamodeling levels. Other definitions of models in the context of model-driven engineering (MDE) are also mentioned in [KWB03, RJB04, Bé05].

Also, many works focus on explaining metamodels. The work in [Kü06] proposes classifying models by its roles in order to support a common understanding of basic notions for MDE such as "model" and "metamodel". In that work there are two kinds of model roles including "token model" and "type model". The work in [AK03] suggests using the "Ontological Metamodeling" approach in addition to the traditional approach "Linguistic Metamodeling" in order to support an essential foundation for model driven development.

Many works focus on the methodology of MDE. The work in [Bé05] proposes a vision of the development of MDE based on lessons learnt in the development of object technology: The principle "Everything is a model" is as an evolution from the principle "Everything is an object". The work in [GSCK04] explains the approach "Software Factories" in which the significant problems of the development of software are discussed. Advanced approaches to them such as language-based, pattern-based, component-based, or transformation-based approaches are explained and integrated towards a model-driven approach for "Software Factories."

Modeling languages play an important role in current model-driven approaches. This point is emphasized in many works [GSCK04, CJKW07]. In these works domain-specific languages are explained as a backbone for model-driven approaches. Defining semantics of modeling languages is still a challenge and a motivation of many works such as the work in [KM08].

The Model Driven Architecture (MDA) is the OMG vision of MDE. The work in [KWB03] presents the original ideas and basic principles of MDA. The work in [Fra03] overviews the OMG vision of MDE together with OMG standards within the context of MDA. The work in [MBMB02] focuses on executable (UML) models towards "programming with models".

Model transformation plays an important role within model-driven approaches. This topic is the focus of the next chapters. For a more detailed view of the Query/Views/Transformation (QVT) standard for model transformation, we recommend to the work in [Kur08].

The model-driven approach in this chapter is based on a foundation of object model that is proposed in [RG98, Ric02]. The work in [EFLR99] is one of the first attempts to precisely specify the semantics of the UML. The authors in [RJB04], who invent the UML, present a full reference to the concepts and constructs of the UML, including their semantics, notations, and purpose. The Object Constraint Language (OCL) is first introduced in [WK98] as an approach for a precise modeling with UML. In [WK03] the authors emphasizes the role of OCL in model-driven approaches in the context of MDA.

The OMG vision of MDE, i.e., MDA [OMG03b] is supported by many different standards such as: MOF [OMG07a], UML [OMG07b, OMG07c], and OCL [OMG06] for modeling and metamodeling; CWM [OMG03a] for the interchange of warehouse and business intelligence metadata in distributed heterogeneous environments; and QVT [OMG07a] for model transformation.

Summary

This chapter has introduced a model-driven approach that covers the OMG vision of MDE. Basic concepts for this approach have been explained. Then, a formal foundation for this approach has been established: Metamodeling and modeling are based on an object-oriented paradigm, and model transformation is based on graph transformation. The next chapter focuses on graph transformation as a formal foundation for model transformation.

Chapter 3

Model Transformation Based on Graph Transformation

This chapter introduces graph transformation as a powerful technique for specifying and applying complex transformations in the context of model-driven approaches. We focus on a special technique based on triple graph grammars (TGGs) and their extensions for transforming models as well as keeping the correspondence between models.

3.1 Introduction

Model transformation can be seen as the heart of model-driven approaches [SK03, CH03] such as model-centric software development approaches (like the UML-based approach) and model-driven engineering (MDE). Within the approaches, graphs are a natural representation for models since most modeling languages are formalized by a visual abstract syntax definition. In this context graph transformation is a promising approach for model transformation, especially, for complex transformations. The advantage of this approach is that graph transformation rules are easily and intuitively specified. In addition, the complexity of graph transformation rules as well as the application formalisms can be hidden for users.

Graph transformation is a mechanism to specify and apply transformations between graphs by so-called graph rewriting rules (graph transformation rules). Graph rewriting is an extension of the well-known string rewriting technique in Chomsky grammars. Informally speaking, a graph transformation rule consists of two graphs corresponding to the left-hand side (LHS) and the right-hand side (RHS) of the rule. For a rule application, nodes and edges in the LHS are matched to certain nodes and edges in the host graph. Then, the matched nodes and edges that do not exist in the RHS are removed from the host graph. The remaining nodes and edges in the RHS are taken as templates for creating new nodes and edges in the host graph. Among differ-

ent approaches to graph transformation such as SPO (Single Pushout) and DPO (Double Pushout), we choose the SPO approach [EHK+97] because the production $P : L \to R$ in SPO is significantly simpler than the one in DPO. Moreover, it is shown in [Lö93] that the SPO is actually a generalisation of the DPO. All existing results with the DPO are still valid with the SPO.

In order to advance in the integration of graph transformation and model-driven approaches, graphs can be extended to attributed graphs for representing models as introduced in [Zie05, HKT02b, GdL06a]. Here, models are seen as object diagrams (in UML). Unlike the approach in [HKT02b, EPT04] in which nodes and edges of attributed graphs are typed by another attributed graph (so called a type graph), within our work nodes and edges of attributed graphs are typed by a Σ-algebra A. In this way types and the semantics of attributed graphs can be defined based on OCL. This allows us to overcome difficulties with typed attributed graphs such as how to explain cardinality, inheritance in meta models, and ordered associations.

Triple graph grammars (TGGs) have been first formally proposed in [Sch95]. Their aim was to ease the description of complex transformations within software engineering. The original TGGs extend earlier ideas from [Pra71]. TGGs allow us to structurally map two graph grammars so that graphs generated by the graph grammars can be related to each other. The mapping between two graph grammars is achieved by inserting a further graph grammar in order to specify the correspondence between their elements. In this way a triple rule is obtained as a composition of three rules with respect to the left, right, and correspondence graph grammars. Then, a triple derivation can be seen as a composition of three derivations corresponding to three graph grammars in the TGGs. Integrated graphs obtained by triple derivations are called triple graphs. From a triple rule we can derive new triple rules for forward and backward transformation, model integration, and model co-evolution. Through these operational scenarios the correspondence between source and target models is established.

In order to promote the integration of TGGs and model-driven approaches, we extend triple graphs to attributed triple graphs like the work in [GdL06a]. Unlike that work in which typed attributed triple graphs are employed, within our approach nodes and edges of attributed triple graphs are typed by a Σ-algebra A. Our aim is also to make use of the expressiveness of OCL. In addition, the OCL-based approach allows us to restrict the correspondence part of triple rules by OCL conditions.

The definition of model transformation based on TGGs has been introduced in [Sch95, EEH08]. Unlike these works in which model transformation is

defined using derived triple rules, we define transformation scenarios based on the relationship between derivations in a triple derivation. In our point of view this is the core of the problem. Moreover, this definition allows us to tackle deleting triple rules. In most current works triple rules are often mentioned as non-deleting rules. We formally specify extensions of TGGs so that TGGs can be applied in the context of QVT. These extensions are often informally mentioned in the current literature.

The rest of this chapter is organized as follows. Section 3.2 presents concepts of graphs and graph morphisms and the extension for attributed graphs. Section 3.3 introduces concepts of graph transformation. Section 3.4 presents triple graph grammars and their extension with attributed triple graphs. Section 3.5 presents the approach based on TGGs for model transformation. Section 3.6 presents extensions of TGGs in order to promote the integration of TGGs and model-driven approaches. This chapter is closed with a summary.

3.2 Graphs and Graph Morphisms

Definition 3.1. (Directed, Labeled Graphs and Morphisms)
Let a set of labels L be given. *A directed, labeled graph* is a tuple $G = (V_G, E_G, s_G, t_G, lv_G, le_G)$, where

- V_G is a finite set of nodes (vertices),
- $E_G \subseteq V_G \times V_G$ is a binary relation describing the edges,
- $s_G, t_G : E_G \to V_G$ are functions that assign a source and a target node to an edge, respectively, and
- $lv_G : V_G \to L$ and $le_G : E_G \to L$ are functions that assign a label to a node and an edge, respectively.

Let two directed, labeled graphs $G = (V_G, E_G, s_G, t_G, lv_G, le_G)$ and $H = (V_H, E_H, s_H, t_H, lv_H, le_H)$ be given. A *graph morphism* $f : G \to H$ is a pair (f_V, f_E), where $f_V : V_G \to V_H$, $f_E : E_G \to E_H$ preserves sources, targets, and labels, i.e., $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$. □

Figure 3.1 shows two directed, labeled graphs G and H and a graph morphism $G \to H$. The mapping is represented by dashed lines between the nodes and edges of G and H.

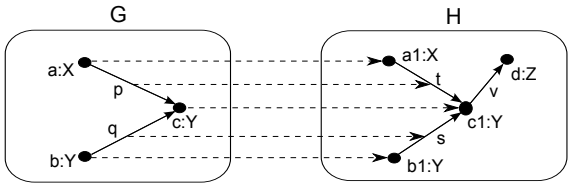


Figure 3.1: Two directed, labeled graphs G and H and a graph morphism $G \to H$. (informally speaking, H 'contains' G)

In order to represent models within model-driven approaches more precisely (as object diagrams within the object-oriented paradigm), graphs have been extended to attributed graphs: Nodes are not only typed but also attributed in order to represent objects with attributes. Edges correspond to links between objects.

Definition 3.2. (Attributed Graphs and Morphisms)
Let a many-sorted signature $\Sigma = (T, OP)$ consisting of a set of sort symbols T, and a family of sets of operations symbols $op : t_1 \times ... \times t_n \to t \in OP$ indexed by their arity be given. Let further a Σ-algebra A such that $|A|$ is the union of the carrier sets A_t of A, for all $t \in T$. An attributed graph (over A) is a pair $AG = (G, A)$ of a graph G and a Σ-algebra A such that $|A| \subseteq V_G$, i.e., the elements of the carrier sets can be included as nodes into the graph.

An attributed graph morphism $f : (G_1, A_1) \to (G_2, A_2)$ is a pair of a Σ-homomorphism $f_A = (f_t)_{t \in T} : A_1 \to A_2$ and a graph homomorphism $f_G = (f_V, f_E) : G_1 \to G_2$ such that $|f_A| \subseteq f_V$, where $|f_A| = \bigcup_{t \in T} f_t$. □

The function f_A maps each value in the carrier sets of A_1 to a value in a corresponding carrier set of A_2. The fact $|f_A| \subseteq f_V$ ensures values of data nodes to be also mapped by f_A when they are mapped by f_V.

Summarizing, data values are represented by nodes which do not have outgoing edges, henceforth called *data-valued nodes* to distinguish them from *object nodes*. Object nodes are linked to data-valued nodes by *attributes*. Edges between object nodes are called *links*. Figure 3.2 pictures an attributed graph in the left part and its representation in a UML-like way in the right part.

Definition 3.3. (Mapping Attributed Graphs with Object Models)
An attribute graph AG conforms to an object model $\mathcal{M}$ iff: (i) they share the same set of types T, and (ii) the labels of attribute edges and link edges of the graph correspond to attribute names and association names of the object model, respectively. □

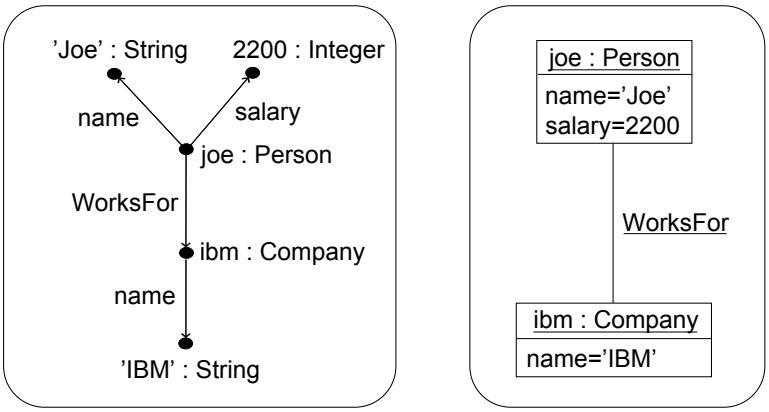


Figure 3.2: An attributed graph in two different notations

3.3 Graph Transformation

Graph transformation is a mechanism to operationally describe the change of graphs through transformation steps by rules. A rule application to a graph is carried out by matching a part of the graph with a pattern and then changing that part by deleting or adding nodes and edges in order to obtain a new graph. This is a natural extension of the technique string rewriting in Chomsky grammars. The production rules on graphs are called graph transformation rules (or graph-rewriting rules).

Definition 3.4. (Graph Transformation Rules)
A graph transformation rule over a Σ-algebra A and a type-indexed family of sets of variables $X = (X_t)_{t \in T}$ is a pair of graphs (L, R), called the LHS and the RHS of the rule, respectively, which are attributed over the term algebra $T_\Sigma(X)$ over Σ and X such that:

- $lv_L(v) = lv_R(v)$ for all $v \in V_L \cap V_R$, and
- $s_L(e) = s_R(e)$, $t_L(e) = t_R(e)$, and $le_L(e) = le_R(e)$ for all $e \in E_L \cap E_R$. $\square$

Figure 3.3 shows an example for graph transformations. The LHS and RHS of the rule are attributed by terms, while the host graph is attributed by constants.

Definition 3.5. (Application of Rules)
An application of a rule $r = (L, R)$ to an attributed graph G (over A) yielding a graph H includes the following steps:

- Choose an occurrence of the LHS L in G by defining an attributed graph morphism $m = (m_G, m_A) : L \to G$, called a *match*, where m_G is a graph homomorphism and the Σ-homomorphism $m_A : T_\Sigma(X) \to A$ becomes the function $I(A, \beta_m)$ for evaluating terms in $T_\Sigma(X)$ with respect to the algebra A and the variable assignment $\beta_m : X \to |A|$.

- Glue the graph G and the RHS R according to the occurrences of L in G so that new items that are presented in R but not in L are added to G. This yields a gluing graph Z. Formally, we have $m(L \cap R) = G \cap Z$ and a homomorphism $R \setminus L \to Z \setminus G$. Here, graphs can be seen as a set of items including vertices and edges.

- Remove the occurrence of L from Z as well as all *dangling edges*, i.e., all edges incident to a removed node. This yields the resulting graph H. Formally, we have $H = Z \setminus m(L)$.

The application of r to G to yield H is called a *direct derivation* from G to H through r and is denoted by $G \overset{r}{\Rightarrow} H$ or simply by $G \Rightarrow H$. □

Figure 3.3 illustrates the steps of a rule application. At the first step the two objects `p:Person` and `c:Company` of the rule together with their attributes are matched with the objects `joe:Person` and `ibm:Company` of the host graph together with their attributes, respectively. According to the RHS of the rule, no objects of the host graph are deleted. The attribute value `2200:Integer` corresponding to `x:Integer` becomes `4400:Integer` which corresponds to `2*x:Integer` in the RHS. In the third step the link `WorksFor` between `joe:Person` and `ibm:Company` in the host graph, which corresponds to the link between `p:Person` and `c:Company` in the LHS is removed because there is no link between the nodes in the RHS.

When the RHS R 'contains' the LHS L, i.e., there is a graph morphism $L \to R$, items of the host graph are not deleted during an application of the rule. The rule is then called a *non-deleting rule*.

Definition 3.6. (Graph Transformation Systems)
A graph transformation system is a structure $GTS = (S, R, A)$ where S is an initial graph and $R = \{r_1, r_2, \ldots, r_n\}$ is a set of graph transformation rules attributed over A. □

Definition 3.7. (Graph Grammars)
A graph grammar is a structure $GG = (S, R, T, A)$ where S is an initial graph, $R = \{r_1, r_2, \ldots, r_n\}$ is a set of graph transformation rules attributed over A, and T is a set of terminal labels.

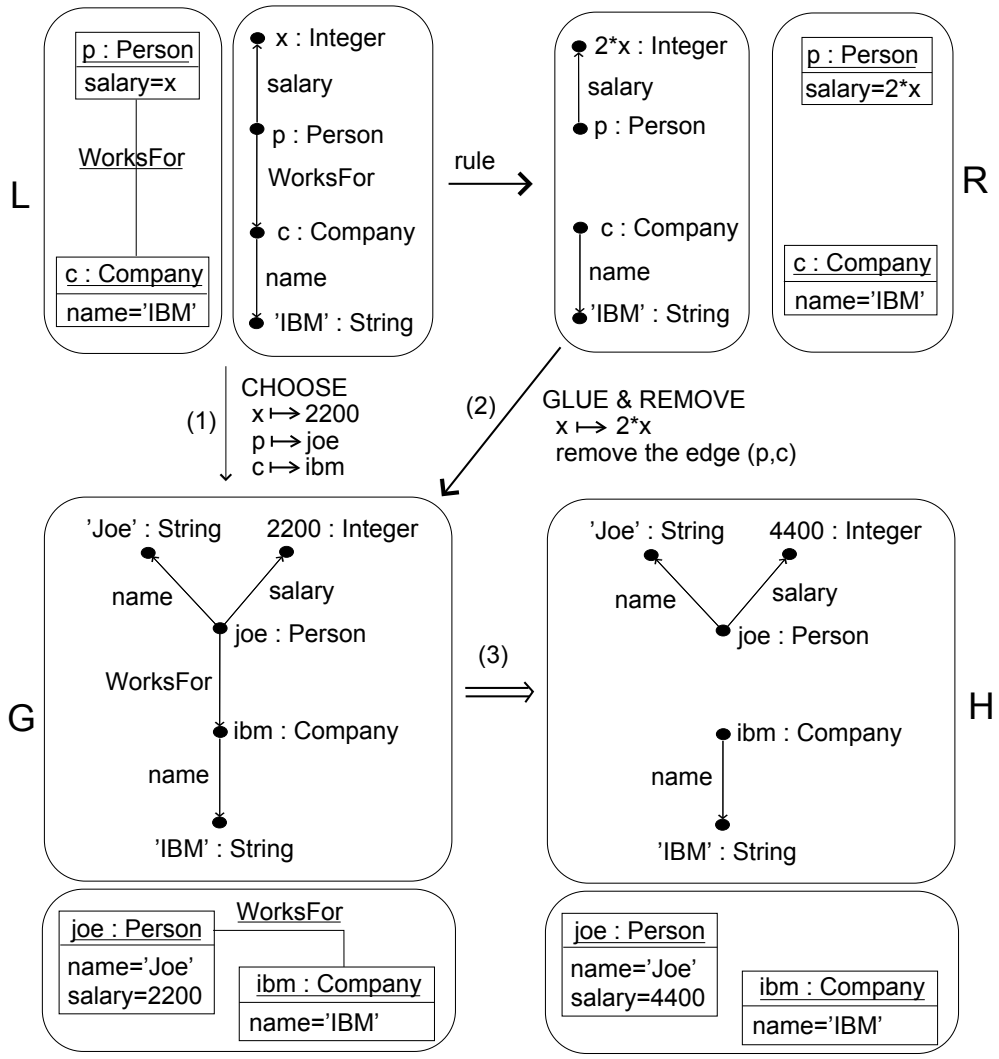


Figure 3.3: Illustration for a graph transformation step

Let a graph G_0 be given. A sequence of direct derivations $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ by rules of R is called a *derivation* from G_0 to G_n by rules of R. The set of all graphs labeled with symbols of T that can be derived from the initial graph S by rules of R, is the *language* generated by the graph grammar GG. □

Definition 3.8. (Model Transformations Based on Graph Transformation)
Let R be a set of rules. A *model transformation* sequence $(G_S, G_S \overset{*}{\Rightarrow} G_T, G_T)$ consists of a source graph G_S, a target graph G_T, and a derivation from G_S to G_T by rules of R. □

3.4 Triple Graph Grammars

TGGs have been proposed as an extension of graph transformation in order to ease the description of complex transformations within software engineering. This formalism offers a declarative specification of bidirectional translations between different graph languages. Specifically, TGGs allow us to structurally map two graph grammars so that graphs generated by the graph grammars can be related to each other. The mapping between two graph grammars is achieved by inserting a further graph grammar in order to specify the correspondence between their elements. In this way a triple rule is obtained as a composition of three rules with respect to the left, right, and correspondence graph grammars. Then, a triple derivation can be seen as a composition of three derivations corresponding to three graph grammars in the TGGs. Integrated graphs obtained by triple derivations are called triple graphs.

Definition 3.9. (Attributed Triple Graphs and Morphisms)
Let Σ-algebra A and three sets T_S, T_T, and T_C of sorts for source, target, and correspondence models, respectively, be given such that $\Sigma = (T, OP)$ and $T = T_S \cup T_C \cup T_T$. An attributed triple graph (over A) is a structure $TAG = (SG \overset{sG}{\leftarrow} CG \overset{tG}{\rightarrow} TG, A)$, where SG, CG, and TG are attributed graphs, called source, correspondence, and target graphs; SG, CG, and TG are attributed over Σ_S, Σ_C, and Σ_T, respectively; $sG : CG \rightarrow SG$ and $tG : CG \rightarrow TG$ are two graph morphisms.

An attributed triple graph morphism $m = (s, c, t) : G \rightarrow H$ between two attributed triple graphs $(SG \overset{sG}{\leftarrow} CG \overset{tG}{\rightarrow} TG, A)$ and $(SH \overset{sH}{\leftarrow} CH \overset{tH}{\rightarrow} TH, A)$ consists of three injective attributed graph morphism $s = (s_A, m_A) : SG \rightarrow$

SH, $c = (c_A, m_A) : CG \to CH$, and $t = (t_A, m_A) : TG \to TH$ such that $s \circ sG = sH \circ c$ and $t \circ tG = tH \circ t$. □

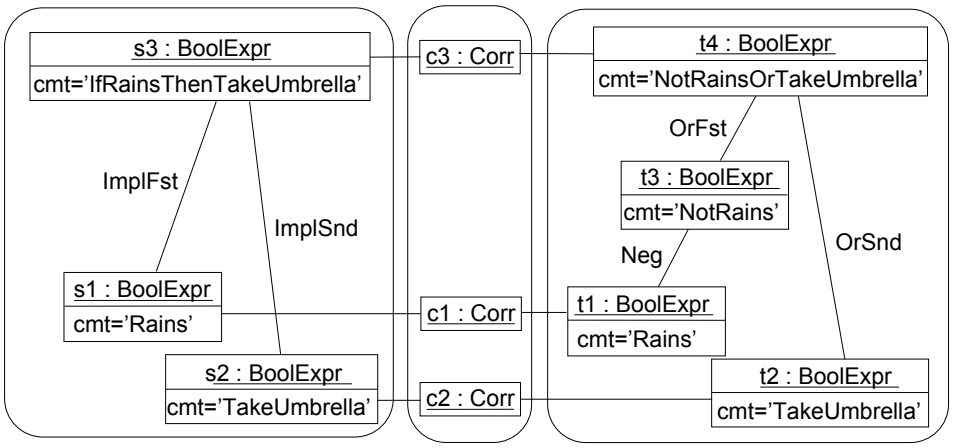


Figure 3.4: An attributed triple graph for a *FLO2TLO* integrated model

Figure 3.4 shows an example attributed triple graph. This graph represents an integrated model of two logic languages. The source language, the so-called *FLO* language (FLO stands for 'Four Logical Operators'), includes logic expressions by four operations $\vee$, $\wedge$, $\neg$, and $\Rightarrow$. The target language, the so-called *TLO* language (TLO stands for 'Three Logical Operators'), includes logic expressions by three logical operations $\vee$, $\wedge$, and $\neg$. A logic expression in *FLO* is logically equivalent to a corresponding logic expression in *TLO*. Such pairs of logic expressions can be represented by an integrated model in form of attributed triple graphs in the so-called *FLO2TLO* language as depicted in Fig. 3.4.

In the left part of the example triple graph, the logic expression $Rains \Rightarrow TakeUmbrella$ is represented by the `BoolExpr` ('Boolean Expression') object $s3$ and two links `ImplFst` ('Implication First') and `ImplSnd` ('Implication Second'). The first link represents the association between the hypothesis $s1$ ($Rains$) and the expression $s3$. The second link represents the association between the conclusion $s2$ ($TakeUmbrella$) and the expression $s3$.

In the right part of the example triple graph, a corresponding logic expression in the target language, $\neg Rains \vee TakeUmbrella$, is shown. This expression is represented by the `BoolExpr` object $t4$ and two links `OrFst` ('Or First') and `OrSnd` ('Or Second') that connect the disjunction to the first and second literals, $\neg Rains$ and $TakeUmbrella$, respectively. The expression $\neg Rains$ is represented by the `BoolExpr` $t3$ and the `Neg` link connecting $t3$ to $t1$.

The `Corr` objects $c1$, $c2$, and $c3$ represent the correspondence between expressions of the source and target models. Here, the graph morphisms sG and tG in the example attributed triple graph are represented by links between the correspondence graph and the source and target graphs.

Definition 3.10. (Triple Rules)
A triple rule over a Σ-algebra A and a type-indexed family of sets of variables $X = (X_t)_{t \in T}$ is a pair of attributed triple graphs (L, R) over the term algebra $T_\Sigma(X)$ over Σ and X, provided with an attributed triple graph morphism $tr : L \to R$ as follows.

$$\begin{array}{ccccccc} L & = & (SL & \xleftarrow{s_L} & CL & \xrightarrow{t_L} & TL) \\ {\scriptstyle tr}\downarrow & & {\scriptstyle s}\downarrow & & {\scriptstyle c}\downarrow & & {\scriptstyle t}\downarrow \\ R & = & (SR & \xleftarrow{s_R} & CR & \xrightarrow{t_R} & TR) \end{array}$$

where (SL, CL, TL) and (SR, CR, TR) are called the LHS and the RHS of the triple rule, respectively. The columns corresponding to (SL, SR) and (TL, TR) are called source and target parts of the rule, respectively. The middle column (CL, CR) is called the correspondence part of the triple rule. □

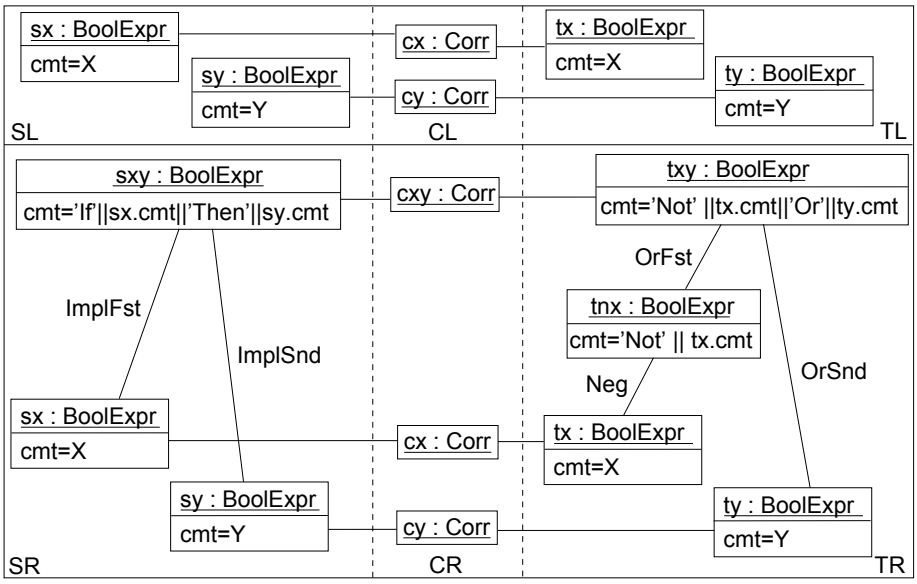


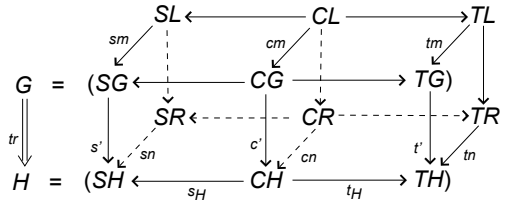
Figure 3.5: A triple rule for a *FLO2TLO* integrated model

According to the definition, the RHS 'contains' the LHS due to the morphism tr. Hence, triple rules are non-deleting rules. Figure 3.5 shows a triple rule for an integrated model of models in *FLO* and *TLO*. The LHS and RHS

of the triple rule are triple graphs corresponding to models in *FLO2TLO* as shown in Fig. 3.4. We can employ OCL expressions to define the value of attributes of object nodes. Here, the 'concat' operator is denoted by '||'.

Definition 3.11. (Application of Triple Rules)
An application of a triple rule $tr = (s, c, t) : L \rightarrow R$ to a given attributed triple graph G to yield a triple graph H consists of the following steps:

- Choose an occurrence of the LHS L in G by defining a triple graph morphism $m = (sm, cm, tm) : L \rightarrow G$, called a *triple match*.
- Apply rules in each part of the rule: $SG \overset{s,sm}{\Longrightarrow} SH$, $CG \overset{c,cm}{\Longrightarrow} CH$, and $TG \overset{t,tm}{\Longrightarrow} TH$. We have the morphism $n = (sn, cn, tn)$, where $sn = SR \rightarrow SH$, $cn = CR \rightarrow CH$, and $tn = TR \rightarrow TH$
- Induce morphisms $s_H : CH \rightarrow SH$ and $t_H : CH \rightarrow TH$ from the morphism n and morphisms in the triple graph $(SR \leftarrow CR \rightarrow TR)$. Therefore, $H = (SH \overset{sH}{\leftarrow} CH \overset{tH}{\rightarrow} TH, A)$ is an attributed triple graph.



The application of tr to G to yield H is called a triple graph transformation step and denoted by $G \overset{tr,m}{\Longrightarrow} H$. □

Figure 3.6 shows an application of the triple rule specified in Fig. 3.5. Once graph morphisms between the parts of triple graphs are represented by links, the application of the triple rule can be seen as the application of a plain rule.

Definition 3.12. (Triple Graph Grammars)
A triple graph grammar is a structure $TGG = (S, TR, TL, A)$, where S is an initial triple graph, $TG = \{r_1, r_2, \ldots, r_n\}$ is a set of triple rules attributed over A, and TL is a set of terminal labels.

The set of all graphs labeled with symbols of TL that can be derived from the initial graph S by triple rules of TR is the *language* generated by the triple graph grammar TGG. □

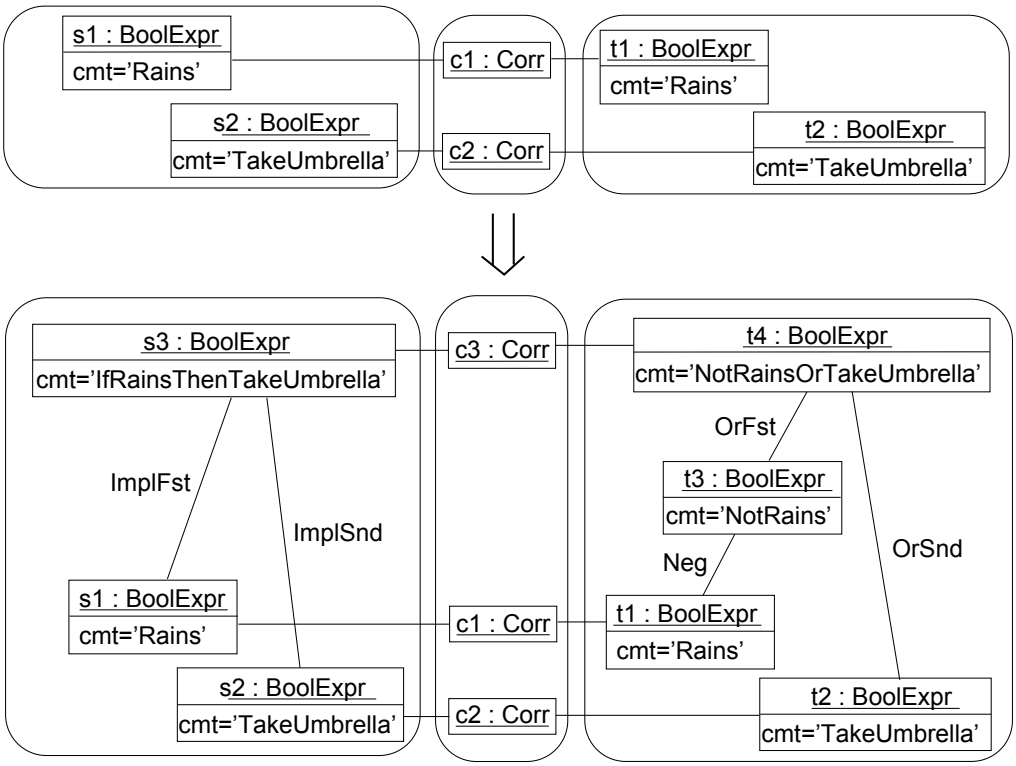


Figure 3.6: Triple transformation step by the triple rule shown in Fig. 3.5 for a *FLO2TLO* integrated model

3.5 Model Transformation Based on TGGs

Structural mappings within triple rules allow us to relate plain derivations with respect to the source, target, and correspondence parts within a triple derivation. Specifically, once a plain rule derivation for the source (or target) model is given, we can induce two derivations corresponding to the remaining parts: The first derivation produces the target (or source) model and the other produces the correspondence between source and target models. In this way operational scenarios of triple rules for model transformations are defined. Formally, the definition of operational scenarios of triple rules are presented as follows.

In the following definitions $TGG = (S, TR, T, A)$ is a triple graph grammar. $VL = (VL_s, VL_c, VL_t)$ is a language generated by the grammar, where VL_s, VL_c, and VL_t are the source, correspondence, and target languages, respectively. They are defined by projecting to the source, correspondence, and target parts of triple graphs in VL, respectively.

Definition 3.13. (Forward Transformation)
Let a graph $G_S \in VL_s$ be given. A forward transformation from G_S to G_T is a computation to define the graph $G_T \in VL_t$ through a triple derivation $S \overset{*}{\Rightarrow} (G_S \leftarrow G_C \rightarrow G_T)$. □

Figure 3.7 shows the context of a forward transformation for a *FLO2TLO* integrated model. We suppose that *FLO2TLO* models are generated by a triple graph grammar $TGG = (S, TR, T, A)$, where S is shown in Fig. 3.7. The source model $G_S \in VL_s$ acts as the input, i.e., the derivation d_S is defined. We need to explicitly define the triple derivation $d_{tr} : S \Rightarrow G = (G_S \leftarrow G_C \rightarrow G_T)$ in order to achieve the triple graph G and subsequently G_T and G_C. In this way a forward transformation from G_S to G_T is realized.

Formally, a forward transformation can be specified as follows. Let $G_S \in VL_s$ together with a derivation d_S: $S_S \overset{s_1,sm_1}{\Longrightarrow} \ldots \overset{s_n,sm_n}{\Longrightarrow} G_S$ be given, where s_i is a part of the triple rule $tr_i = (s_i, c_i, t_i)$. From the initial triple graph S and the derivation d_S, we need to define triple matches $m_i = (sm_i, cm_i, tm_i)$ in order to explicitly define the derivation $d_{tr} : S \overset{tr_1,m_1}{\Longrightarrow} \ldots \overset{tr_n,m_n}{\Longrightarrow} G = (G_S \leftarrow G_C \rightarrow G_T)$ and subsequently the derivation $d_T : S_T \overset{*}{\Rightarrow} G_T$. Through the derivation d_T the target model G_T can be defined. Note that the determinism of a forward transformation depends on the determinism of triple matches $m_i = (sm_i, cm_i, tm_i)$ induced from sm_i.

Definition 3.14. (Backward Transformation)
Let a graph $G_T \in VL_t$ be given. A backward transformation from G_T to

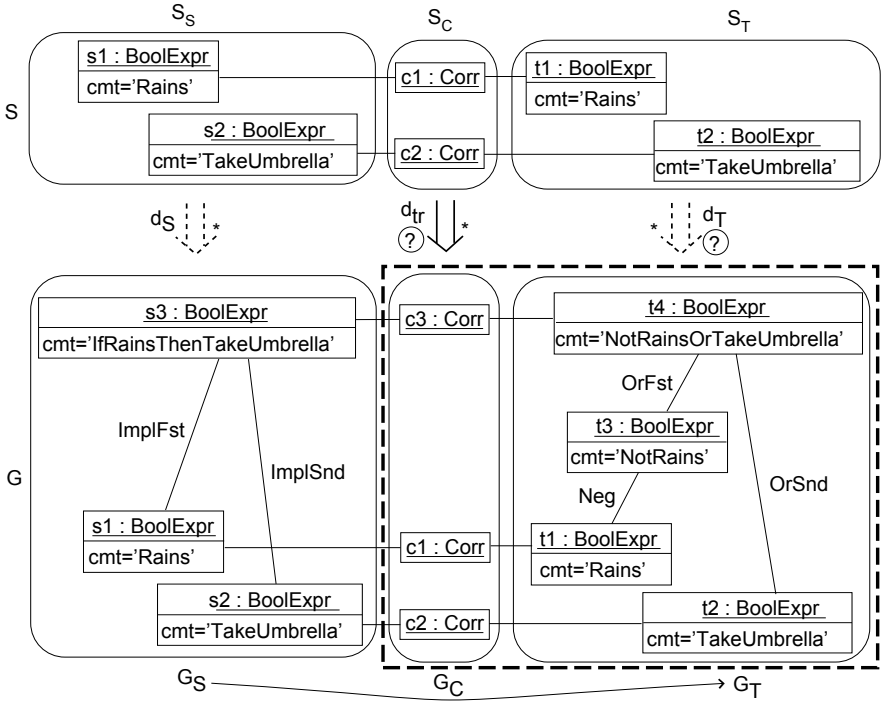


Figure 3.7: A forward transformation from G_S to G_T for a *FLO2TLO* integrated model

G_S is a computation to define the graph $G_S \in VL_s$ through a derivation $S \overset{*}{\Rightarrow} (G_S \leftarrow G_C \rightarrow G_T)$.□

Figure 3.8 shows the context of a backward transformation for a *FLO2TLO* integrated model. The target model $G_T \in VL_t$ is given, i.e., the derivation d_T can be defined. We need to explicitly define the triple derivation d_{tr} : $S \Rightarrow G = (G_S \leftarrow G_C \rightarrow G_T)$ in order to achieve the triple graph G and subsequently G_S and G_C. In this way a backward transformation from G_T to G_S is realized.

Definition 3.15. (Model Integration)
Let the graphs $G_S \in VL_t$ and $G_T \in VL_s$ be given. A model integration of G_S and G_T is a computation to define a derivation $S \overset{*}{\Rightarrow} (G_S \leftarrow G_C \rightarrow G_T)$.
□

Figure 3.9 shows the context of a model integration for a *FLO2TLO* integrated model. The target and source models $G_S \in VL_s$ and $G_T \in VL_t$ are given, i.e., the derivations d_S and d_T can be defined. We need to explicitly define the triple derivation $d_{tr} : S \Rightarrow G = (G_S \leftarrow G_C \rightarrow G_T)$ in order to achieve the triple graph G and subsequently G_C for the correspondence between G_S and G_T. In this way a model integration of G_S and G_T is realized.

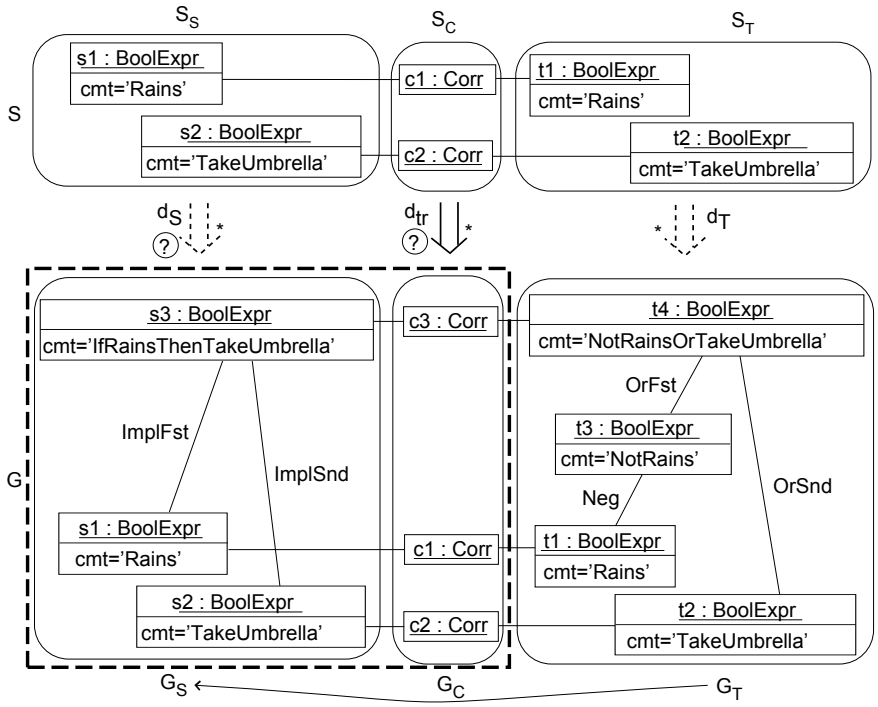


Figure 3.8: A backward transformation from G_T to G_S for a *FLO2TLO* integrated model

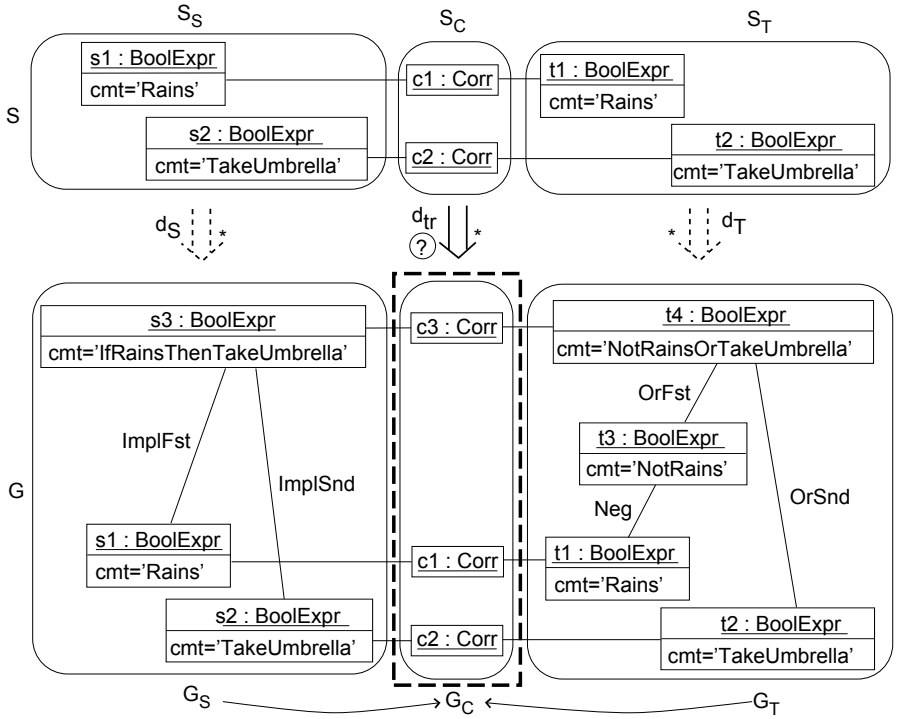


Figure 3.9: A model integration of G_S and G_T for an integrated *FLO2TLO* model

Definition 3.16. (Model Co-Evolution)
Let $E_S \in VL_s$ and $E_T \in VL_t$ be graphs as source and target parts of a triple graph E, respectively. A model co-evolution from (E_S,E_T) to (F_S,F_T) is a computation to define graphs $F_S \in VL_s$ and $F_T \in VL_t$ through the derivation $(E_S \leftarrow E_C \rightarrow E_T) \overset{*}{\Rightarrow} (F_S \leftarrow F_C \rightarrow F_T)$. □

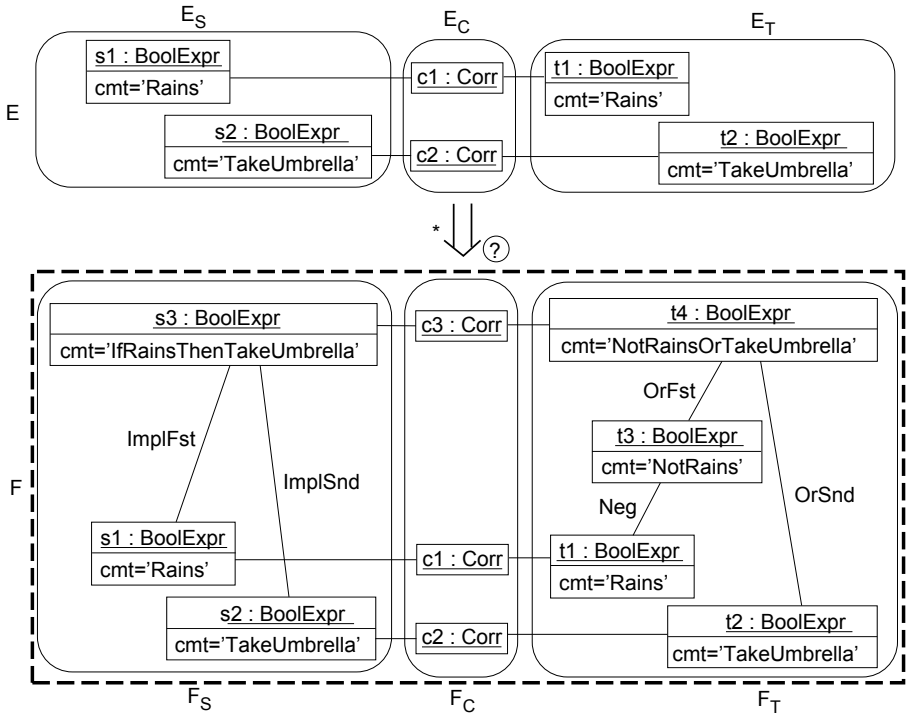


Figure 3.10: A model co-evolution from E_S and E_T to F_S and F_T for an integrated *FLO2TLO* model

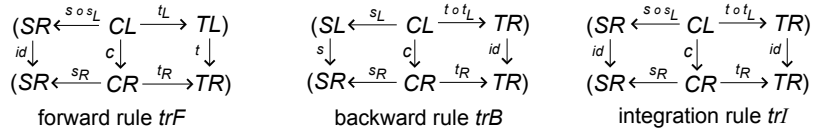
Figure 3.10 shows an example for model co-evolution. The pair of models E_S and E_T corresponding to a *FLO2TLO* integrated model is transformed to the pair of models F_S and F_T for a new *FLO2TLO* integrated model. This model co-evolution is realized by a triple derivation $d_{tr} : E = (E_S \leftarrow E_C \rightarrow E_T) \overset{*}{\Rightarrow} F = (F_S \leftarrow F_C \rightarrow F_T)$

Summarizing, the basic idea of the above operational scenarios is that we need to define triple matches $m_i = (sm_i, cm_i, tm_i)$ using correspondences in the rule tr_i together with the match sm_i, tm_i, or (sm_i, tm_i) with respect to the derivation d_S, d_T, or (d_S, d_T) for the forward transformation, backward transformation, or model integration, respectively.

Since triple rules are non-deleting rules, the input model (G_S or G_T) retains images of matches within the triple derivation, e.g., $sm_i \subset sm_n$ for $i = \overline{1,n}$.

Therefore, in principle triple derivations for transformation scenarios can start with the input model instead of the initial graph S. This is the basic idea of derived triple rules.

Definition 3.17. (Derived Triple Rules)
Each triple rule $tr = L \to R$ derives forward, backward, and integration rules as follows:



where id is the identify function. In each derived rule there is a part of the rule in which the LHS coincides with the RHS. □

The resulting model G_T^F of a forward transformation can be defined by a derivation with forward rules as follows.

$(G_S \leftarrow S_C \rightarrow S_T) \overset{trF_1,m_1}{\Longrightarrow} \ldots \overset{trF_n,m_n}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T^F)$,

where trF_i is the forward rule derived from tr_i.

The derivation with backward rules for a backward transformation is defined as follows.

$(S_S \leftarrow S_C \rightarrow G_T) \overset{trB_1,m_1}{\Longrightarrow} \ldots \overset{trB_n,m_n}{\Longrightarrow} (G_S^B \leftarrow G_C \rightarrow G_T)$,

where trB_i is the backward rule derived from tr_i.

The derivation with integration rules is defined as follows.

$(G_S \leftarrow S_C \rightarrow G_T) \overset{trI_1,m_1}{\Longrightarrow} \ldots \overset{trI_n,m_n}{\Longrightarrow} (G_S \leftarrow G_C^I \rightarrow G_T)$,

where trI_i is the forward rule derived from tr_i.

Based on the mechanism of triple rule applications, we can prove that $G_T = G_T^F$, $G_S = G_S^B$, and $G_C = G_C^I$ with respect to forward transformation, backward transformation, and model integration. For a detailed proof, we refer the reader to [Sch95, EEH08]. In other words, the resulting model by derived rules coincides with the one induced from definitions of operational scenarios.

3.6 Extensions of Triple Graph Grammars

In order to promote the integration of TGGs and model-driven approaches, especially in the context of QVT, TGGs need to be extended. In this section we discuss two extensions of TGGs and formally explain why TGG features of model transformation can be kept in the extensions. The first extension which is often informally mentioned in the current literature is about the connection between the correspondence part and the source or target part. The second extension is mentioned in our previous work [DG09a] in which triple rules can be deleting rules.

3.6.1 View Triple Graphs as Plain Graphs

In this extension a node in the correspondence part of a triple rule can be connected to many nodes having different types in source and target parts. Specifically, a correspondence node can access nodes in source and target parts via navigation operations (with links and role names). This extension allows us to view triple graphs as plain graphs. In this extension morphisms in a triple graph are equivalent to navigation operations in object diagrams.

Definition 3.18. (Extended Attributed Triple Graphs)
Let a Σ-algebra A and three sets T_S, T_T, and T_C of sorts for source, target, and correspondence models be given, where $\Sigma = (T, OP)$ and $T = T_S \cup T_C \cup T_T$. An attributed triple graph (over A) is a structure $TAG = (SG \overset{sG}{\leftarrow} CG \overset{tG}{\rightarrow} TG, A)$, where SG, CG, and TG are attributed graphs, called source, correspondence, and target graphs; SG, CG, and TG are attributed over Σ_S, Σ_C, and Σ_T, respectively; $sG : CG \times T_S \rightarrow SG$ and $tG : CG \times T_T \rightarrow TG$ are two graph morphisms.

An extended attributed triple graph morphism $m = (s, c, t) : G \rightarrow H$ between two extended attributed triple graphs $(SG \overset{sG}{\leftarrow} CG \overset{tG}{\rightarrow} TG, A)$ and $(SH \overset{sH}{\leftarrow} CH \overset{tH}{\rightarrow} TH, A)$ consists of three injective attributed graph morphism $s = (s_A, m_A) : SG \rightarrow SH$, $c = (c_A, m_A) : CG \rightarrow CH$ and $t = (t_A, m_A) : TG \rightarrow TH$ such as $s(sG(v_{CG}, T_S)) = sH(c(v_{CG}), T_S)$ and $t(tG(v_{CG}, T_T)) = tH(c(v_{CG}, T_T))$ for all nodes v_{CG} in CG. □

Figure 3.11 shows an extended attributed graph in which a node, e.g., the correspondence node `c` can connect to more than one node in the source or target part. The extension does not change the definitions of triple derivations so that the TGG features of model transformation can be retained.

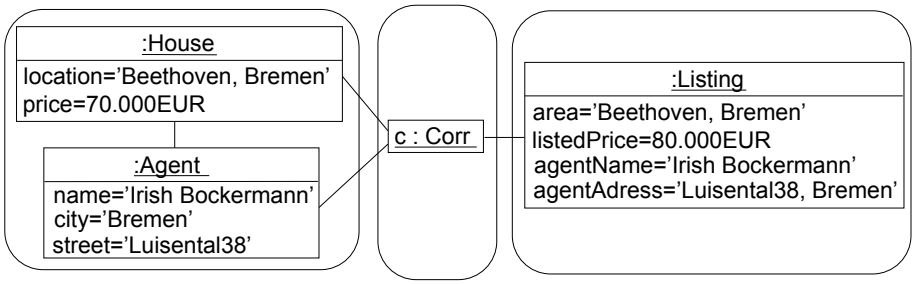


Figure 3.11: A correspondence node may be linked to many nodes in the source or target part of a triple graph

3.6.2 Deleting Triple Rules

TGGs have been defined as non-deleting rules [Sch95], i.e., the LHS is included in the RHS of a triple rule. This restriction is necessary for derived triple rules to be applicable for operational scenarios of triple rules. Within the context of QVT, in many cases model elements can be deleted by transformations. So triple rules should be extended to deleting rules.

Triple rules derived from a triple rule can realize operational scenarios of the triple rule only if the input model maintains the information of its matches with any derivation creating the input model (source or target models). In general, deleting triple rules do not fulfill the feature. The applicability of derived triple rules which are derived from a deleting triple rule can be summarized as follows.

- Derived triple rules are applicable for model co-evolution in any cases, i.e., the rules can be deleting or non-deleting rules.
- The derived triple rule for the back or forward transformation is applicable only if the plain rule corresponding the target or source part of triple rules are non-deleting rules, respectively.
- In the remaining cases derived triple rules are inapplicable.

Summary

This chapter has introduced and complemented concepts, principles, and extensions of graph transformation and triple graph grammar, supporting a foundation for model transformation. Models can be represented by attributed graphs: Nodes are not only typed but also attributed by a Σ-algebra A. An attributed graph conforms to an object model $\mathcal{M}$ only if it is typed by the type system of $\mathcal{M}$ and attribute values are defined over the illustration of $\mathcal{M}$. We have shown that attributed graphs, graph morphisms, and graph grammars can be extended to attributed triple graphs, triple graph morphisms, and triple graph grammars, respectively.

This chapter has explained how model transformation can be realized by plain rules and triple rules. Model transformation based on TGGs allows us to keep the trace between source and target models. Operational scenarios of a triple rule can be realized by corresponding derived triple rules. The operational scenarios include forward transformation, backward transformation, model integration, and model co-evolution. In order to promote the integration of TGGs in model-driven approaches, TGGs have been extended in several ways: Triple graphs can be seen as plain graphs, and triple rules can be deleting rules. Incorporating OCL in TGGs is a further extension of TGGs for this aim, which is the focus of the next chapter.

Chapter 4

Incorporating OCL in TGGs

This chapter explains an extension of TGGs in which the expressiveness of triple rules for complex transformations within model-driven approaches is enhanced. We use OCL conditions to establish restrictions on the applicability of triple rules and derived triple rules. The incorporation of OCL in TGGs offers a declarative description of transformation relationships and the correspondence between source and target models, which can be compared to the QVT Relational language. We propose a language called USE4TGG in order to present the declarative description. This language is the key for realizing model transformation based on the integration of TGGs and OCL in a UML-based metamodeling framework.

4.1 Introduction

Explaining the relationship between models, especially transformation relationships is the heart of model-driven approaches such as MDE. Up to now, the main results of model-driven approaches are based on the object-oriented paradigm under the support of UML as a key enabling technology. Within the OMG vision of MDE, QVT (Queries, Views, Transformations) has been proposed as the standard for model transformation. Well-founded approaches to model transformation like TGGs are particularly of interest in the context of QVT. TGGs and QVT share many functions and building blocks [GK07]. But in contrast to TGGs, QVT includes the declarative language OCL (Object Constraint Language) which allows us to express properties and to navigate in complex models. We propose an incorporation of OCL in TGGs towards a declarative description for model transformation like the QVT Relational language. Our goal is to increase the expressiveness of TGGs as well as to promote the integration of TGGs in model-driven approaches.

In the recent literature, a need for an integration of TGGs and OCL has been expressed. However, such an integration has not been systematically studied or realized yet. Recent papers [KS04, Kö05, KS06, KW07, GdL06b] often

employ OCL only for updating attribute expressions. Using OCL conditions as restrictions on triple rules is not mentioned.

Giving restrictions on transformation rules is an approach for increasing the expressiveness of rules. In [EH86, Kre93, HHT96] context conditions of rule applications, called application conditions (ACs), were introduced. Application conditions require or forbid the existence of nodes, edges, or certain subgraphs in the given graph. They are called positive or negative application conditions, respectively. They also can be classified into application pre- and postconditions. Preconditions (left-sided constraints) are used to restrict possible matches of the rule application. Postconditions (right-sided constraints) are used to check effects of the rule application. Those works present application conditions as graphs. The work in [ZHG05] proposes using OCL conditions as application preconditions. OCL application conditions are formed by OCL formulas including attribute expressions and navigation expressions. The work in [CCGdL08] shows that negative application conditions can be re-expressed by OCL conditions.

We propose using OCL conditions as application conditions for triple rules. Due to the extension of triple graphs to triple attributed graphs as mentioned in Chapter 3, we can embody OCL conditions into triple rules. We employ OCL conditions not only as application preconditions but also as application postconditions of triple rules. We also support OCL metamodel invariants, especially for correspondence parts (when we need the automatic construction of metamodels for correspondence parts) in triple rules. We show that application conditions for derived triple rules can also be obtained from the triple rule and its application conditions.

We propose the USE4TGG language for the integration of TGGs and OCL: One part of this language has a one-one mapping to TGGs and the remaining part covers OCL concepts. Like the QVT Relational language, this language offers a declarative description for model transformation. This language also allows us to present the incorporation of OCL in TGGs as well as to realize it within a UML metamodeling framework like the UML-based Specification Environment (USE) [GBR07].

The rest of this chapter is structured as follows. Section 4.2 explains the basic idea by means of an example. Section 4.3 focuses on OCL ACs of triple rules. Section 4.4 presents OCL ACs of derived triple rules. Section 4.5 presents the language USE4TGG. Section 4.6 comments on related work. The chapter is closed with a summary.

4.2 Basic Idea

This section explains the basic idea of our approach by means of an example. We consider a typical situation within software development, where one wants to formally trace the relationship between a conceptual model and a design model. Figure 4.1 indicates the correspondence between an association end on the left side within a conceptual model and an attribute and an operation on the right side within a design model. For example, we recognize that the association end `customer` is mapped to the attribute `customer` and the operation `getCustomer()` of the class `Order`.

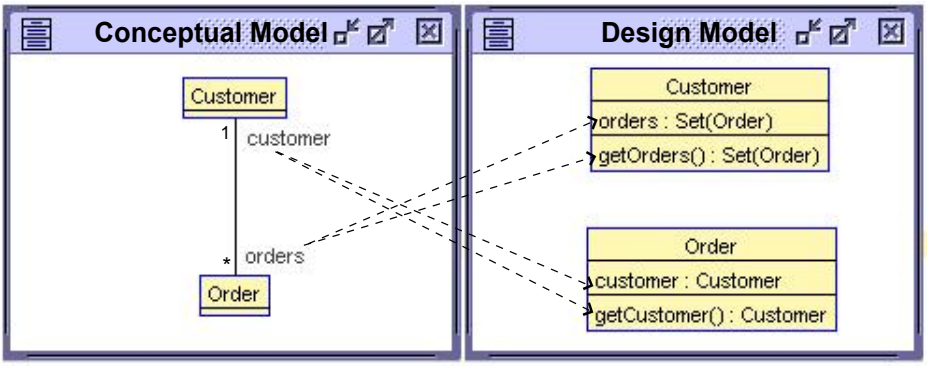


Figure 4.1: Correspondence between association ends and attributes

We want to express the requirement that, when we add an association between `Customer` and `Order` in the conceptual model, the corresponding classes in the design model are automatically extended with new attributes and operations. Therefore, a correspondence between association ends in the conceptual model and the attributes and operations in the design model has to be established in a formal way. This scenario is a model transformation between UML class diagrams. In the following we point out that QVT as well as TGGs incorporating OCL can approach these requirements.

Figure 4.2 shows the basis of the transformation in a schematic form as a metamodel. The left side shows the metamodel for the conceptual domain, the right side shows the metamodel for the design domain, and the middle part shows the connection for a correspondence between them. Later we will see that the diagram in Fig. 4.1 can be seen as an instance diagram for the metamodels and that the dashed arrows are represented as objects belonging to classes from the middle part. A TGG rule incorporating OCL as well as QVT mappings can realize an action which adds an association and updates the attributes and operations as explained in the following.

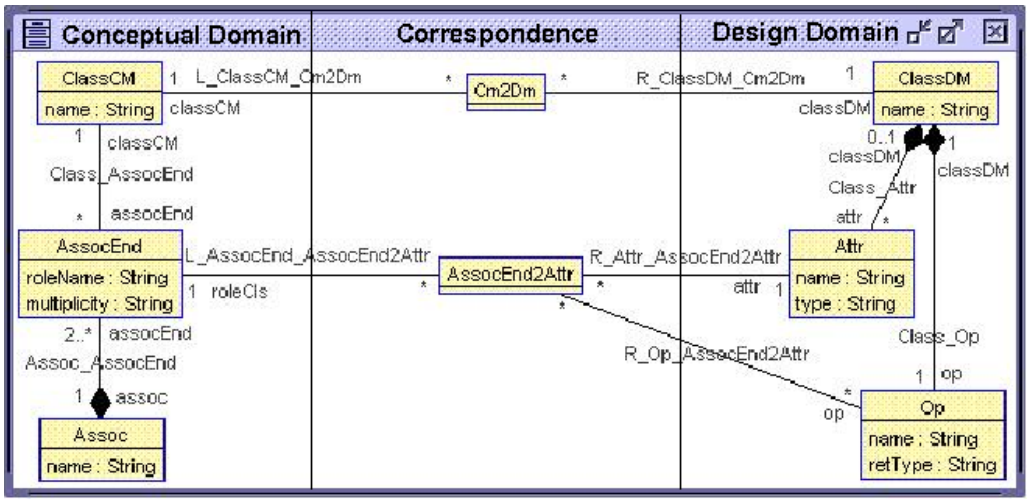


Figure 4.2: Simplified example metamodels and their connection

4.2.1 QVT for the Example Transformation

Figure 4.3 presents the example transformation in the QVT Core language. We choose the QVT Core language rather than the QVT Relation language because on the one hand its expressiveness is as robust as the QVT Relation language, and on the other hand the QVT Relation language can be transformed to the QVT Core language.

The example transformation contains two mappings `cm2Dm` and `assocEnd2Attr`. The mappings declare constraints that must hold between the model elements within the transformation. The transformation requires two new types of elements (classes) `Cm2Dm` and `AssocEnd2Attr` for the correspondence domain. The mappings can be executed by matching patterns which are declared within the '`check`' parts with respect to domain areas, and the '`where`' part with respect to middle area. Each area of a mapping includes a guard pattern and a bottom pattern as pre- and postconditions of the mapping, respectively.

The `cm2Dm` mapping indicates the correspondence between pairs of objects of the conceptual and design models, which coincide in their name. The `assocEnd2Attr` mapping is fulfilled only if:

- Patterns of the domain `cm` in the first '`check`' part are matched, i.e., two `ClassCM` objects in the guard pattern together with three objects in the bottom part must fulfil the OCL constraints in the bottom pattern.
- Patterns of the domain `dm` in the second '`check`' part are matched, i.e.,

two objects of `ClassDM` for the guard pattern are accessed; afterwards two `Attr` objects and two `Op` objects are created and their attribute values are updated by the assignments in the bottom part.

- Patterns of the middle area in the '`where`' part are matched, i.e., two objects `oneMapping` and `manyMapping` fulfill the constraints of the `cm2Dm` mapping.

```
class Cm2Dm{
 classCM: ClassCM;
 classDM: ClassDM;
}


class AssocEnd2Attr{
 role: AssocEnd;
 op:Op;
 attr: Attr;
}
-----------------------

map cm2Dm{
 check cm(){
  classCM: ClassCM;
 }
 check dm(){
  classDM: ClassDM;
 }
 where(){cm2dm:Cm2Dm |
  cm2dm.classCM=classCM;
  cm2dm.classDM=classDM;
  classCM.name
          =classDM.name;
 }
}
```

```
map assocEnd2Attr{
 check cm(oneCM:ClassCM; manyCM:ClassCM){
  oneRole:AssocEnd,assoc:Assoc,manyRole:AssocEnd|
  oneRole.class=oneCM; oneRole.assoc=assoc;
  manyRole.class=manyCM; manyRole.assoc=assoc;
  oneRole.multiplicity='1'; manyRole.multplc='*';
  oneRole.roleName<>oclUndefined(String);
  manyRole.roleName<>oclUndefined(String);}
 check enforce dm(oneDM:ClassDM;manyDM:ClassDM){
  realize oneAttr:Attr, realize manyAttr:Attr,
  realize oneOp:Op, realize manyOp:Op |
  oneAttr.class:=manyDM; manyAttr.class:=oneDM;
  oneOp.class:=manyDM; manyOp.class:=oneDM;
  oneAttr.type:=oneDM.name;
  oneOp.retType:=oneDM.name;
  manyAttr.type:=
        'Set('.concat(manyDM.name).concat(')');
  manyOp.retType:=manyAttr.type;}
 where(oneMapping:Cm2Dm, manyMapping:Cm2Dm){
  realize one2Attr:AssocEnd2Attr,
  realize many2Attr:AssocEnd2Attr |
  one2Attr.role:=oneRole;one2Attr.attr:=oneAttr;
  one2Attr.op:=oneOp; many2Attr.role:=manyRole;
  many2Attr.attr:=manyAttr;many2Attr.op:=manyOp;
  oneAttr.name:=oneRole.roleName;
  oneOp.name:='get'.concat(oneRole.roleName);
  manyAttr.name:=manyRole.roleName;
  manyOp.name:='get'.concat(manyRole.roleName)
 }
}
```

Figure 4.3: Example Transformation in the QVT Core language.

The keyword '`enforce`' in the QVT Core language means that in case the OCL constraints within the bottom pattern do not hold, the target model, i.e., the design model in this case must be changed to fulfil the constraints. The keyword '`realize`' means that when the bottom pattern is in the enforce mode, variables attached with '`realize`' (the so-called realized variables) must bind to values.

The transformation executes when its mappings are fulfilled. A combination of the mappings `cm2Dm` and `assocEnd2Attr` fulfils the requirements stated for the example transformation `addAssociation`.

4.2.2 TGGs and OCL for the Example Transformation

Figure 4.4 shows the triple rule `addAssociation` incorporating OCL in a QVT-like style. The LHS corresponds to guard patterns, and the RHS corresponds bottom patterns. The LHS is included in the RHS (i.e., the rule is a non-deleting rule).

OCL constraints of the triple rule in Fig. 4.4 are presented on the top and bottom of the LHS and RHS, respectively. The OCL constraints cannot only be employed to restrict the attribute values in guard and bottom patterns, but they can also express more general restrictions on graphs as object diagrams.

Classical TGG rules do not embody OCL constraints. Our approach allows us to add OCL constraints in all parts and thus enables a general form for restricting the rule application as well as a fine-grained description of consequences of rule applications, for example, to describe the updating of object attributes. General OCL expressions, which can traverse the complete working graph on which the rule is applied, are allowed as right sides of attribute assignments. The evaluation of the OCL expressions is made before the working graph is modified by the rule.

The triple rule as shown in Fig. 4.4 works as follows. When it is applied, an `Assoc` object is created for representing an association in the conceptual domain. This results in a graph rewrite in the design domain, i.e., `Attr` objects and `Op` objects representing the association are created. The correspondence between the added association and the new attributes and operations is established by a graph rewrite in the correspondence domain. The OCL constraints allow us to check the applicability of the rule and to update the attribute values within the design domain.

4.2.3 Requirements to Incorporate OCL in TGGs

By considering QVT as a motivation for the incorporation of OCL in TGGs as well as by generalizing results from subsections 4.2.1 and 4.2.2, we propose requirements for incorporating OCL in TGGs as follows.

- Supporting OCL formulas for attribute expressions.
- Supporting OCL pre- and postconditions of triple rules.
- Supporting constraints on the metamodel for the correspondence part

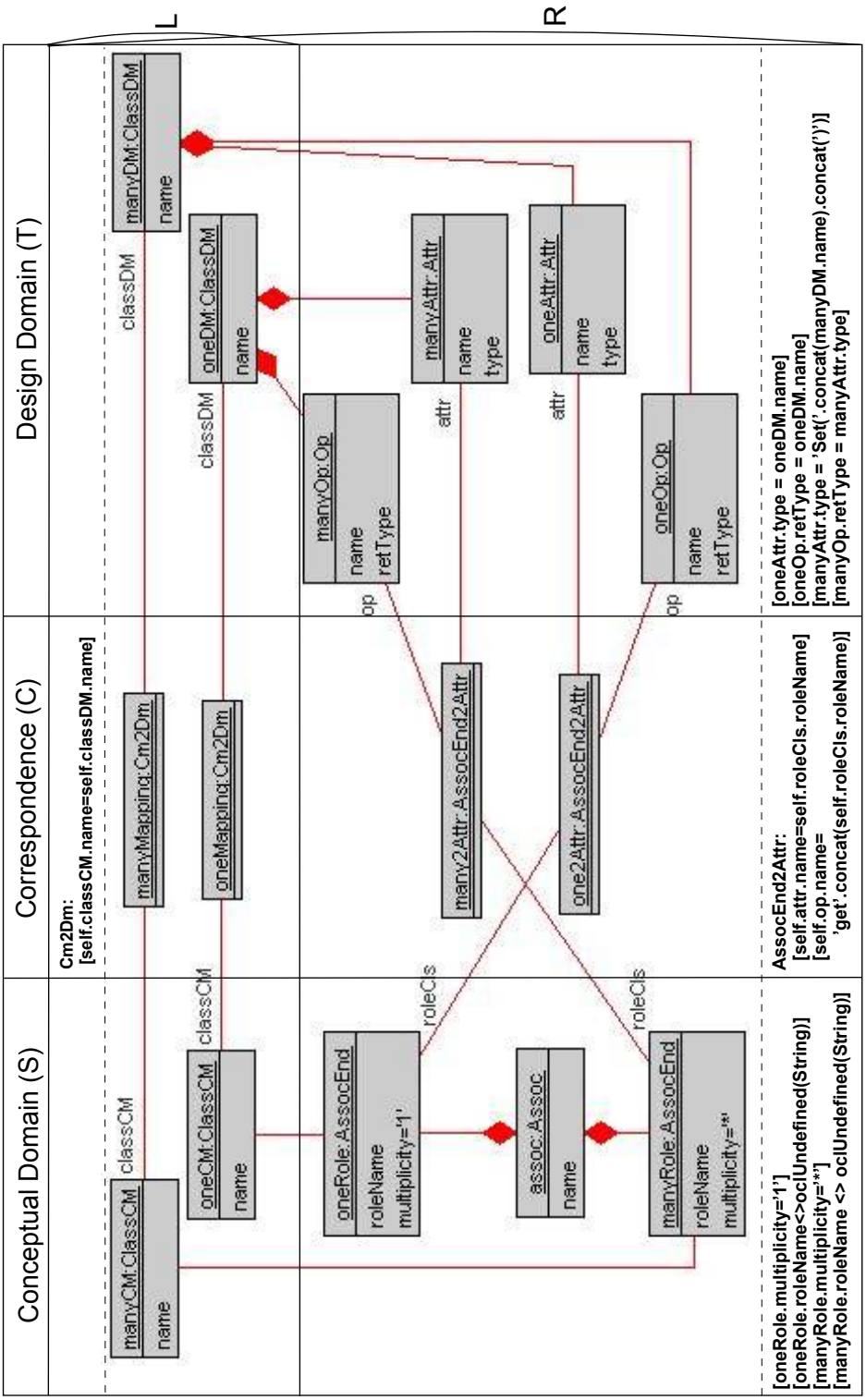


Figure 4.4: The triple rule `addAssociation` incorporating OCL.

of TGG rules. We suppose that all of the constraints must be taken from TGG rules.

In addition, the integration of TGGs and OCL requires variations of TGGs. A node in the correspondence domain may be connected to multiple nodes in the remaining domains. This is a variation of TGGs because in the original TGG definition [Sch95], the mappings are only one-one. Within our extension, TGG rules can be deleting rules whereas in their original definition, TGG rules are non-deleting rules. These variations have been tackled as explained in Chapter 3.

4.3 OCL Conditions for Triple Rules

In this section we specify formally OCL application conditions of triple rules. Since restrictions in the source and target parts of a triple rule are independently defined, application conditions of a triple rule can be defined as a combination of pre- and postconditions in the source, target, and correspondence parts of the triple rule.

4.3.1 OCL Conditions for Source and Target Parts

OCL conditions for the source or target part of a triple rule are OCL application conditions of a plain rule that corresponds to this part. We can use OCL conditions to restrict a rule, because on the one hand, the LHS and RHS of the rule are attributed graphs. On the other hand, according to Chapter 3, nodes of attributed graphs that conform to an object model $\mathcal{M}$ coincide with OCL terms. This allows us to use OCL conditions as constraints on attributed graphs.

Definition 4.1. (OCL Constraints on Attributed Graphs)
An OCL constraint e on an attributed graph is a boolean OCL term such that it can be evaluated within any illustration $I(\sigma, \beta)$ of the graph. An attributed graph (does not) satisfies an OCL constraint in an illustration iff the evaluation of the constraint in the illustration is (false) true. □

Definition 4.2. (OCL Application Conditions)
OCL application conditions (BACs)[1] of a rule are OCL constraints on attributed graphs corresponding to the LHS and RHS of the rule. BACs in the LHS and RHS are pre- and postconditions of the rule, respectively. □

[1]BACs stands for Boolean Application Conditions

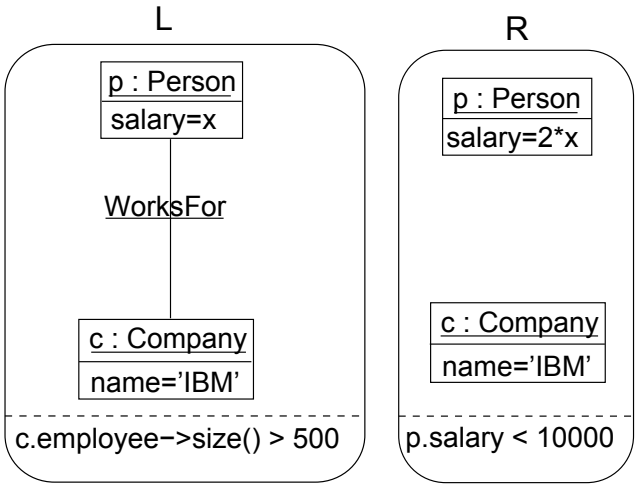


Figure 4.5: An example of OCL application conditions for plain rules.

Figure 4.5 shows the OCL application precondition for the example rule: `comp.employee→size()>500`. It means that the rule is applicable if the company `c:Company` employs more than 500 employees. The OCL postcondition `p.salary<10000` means that the rule is applied in a correct way only if the salary of the person p:Person is less than 10000.

Definition 4.3. (Application Condition Fulfillment)
A transformation rule with BACs is a tuple $r = (L, R, BAC)$, where BAC includes OCL application conditions.

A graph H is derived from a graph G by a rule $r = (L, R, BAC)$ and a match m iff:

- H is derived from G by $(L \rightarrow R, m)$,
- the LHS satisfies all preconditions in BAC within $I(\sigma_G, \beta_m)$, and
- the RHS satisfies all postconditions in BAC within $I(\sigma_H, \beta_{r,m})$,

where σ_G and σ_H are models derived from G and H, respectively; β_m is the variable assignment part of the match m, and $\beta_{r,m}$ is the variable assignment part for the RHS from H within the rule application. □

Let's check whether the rule in Fig. 4.5 is applicable for the transformation shown in Fig. 3.3. To check the application preconditions of this rule means to check whether the company `ibm` employees more than 500 employees. When the rule is applied, to check the application postconditions means to check whether the `salary` of `joe` is less than 10000.

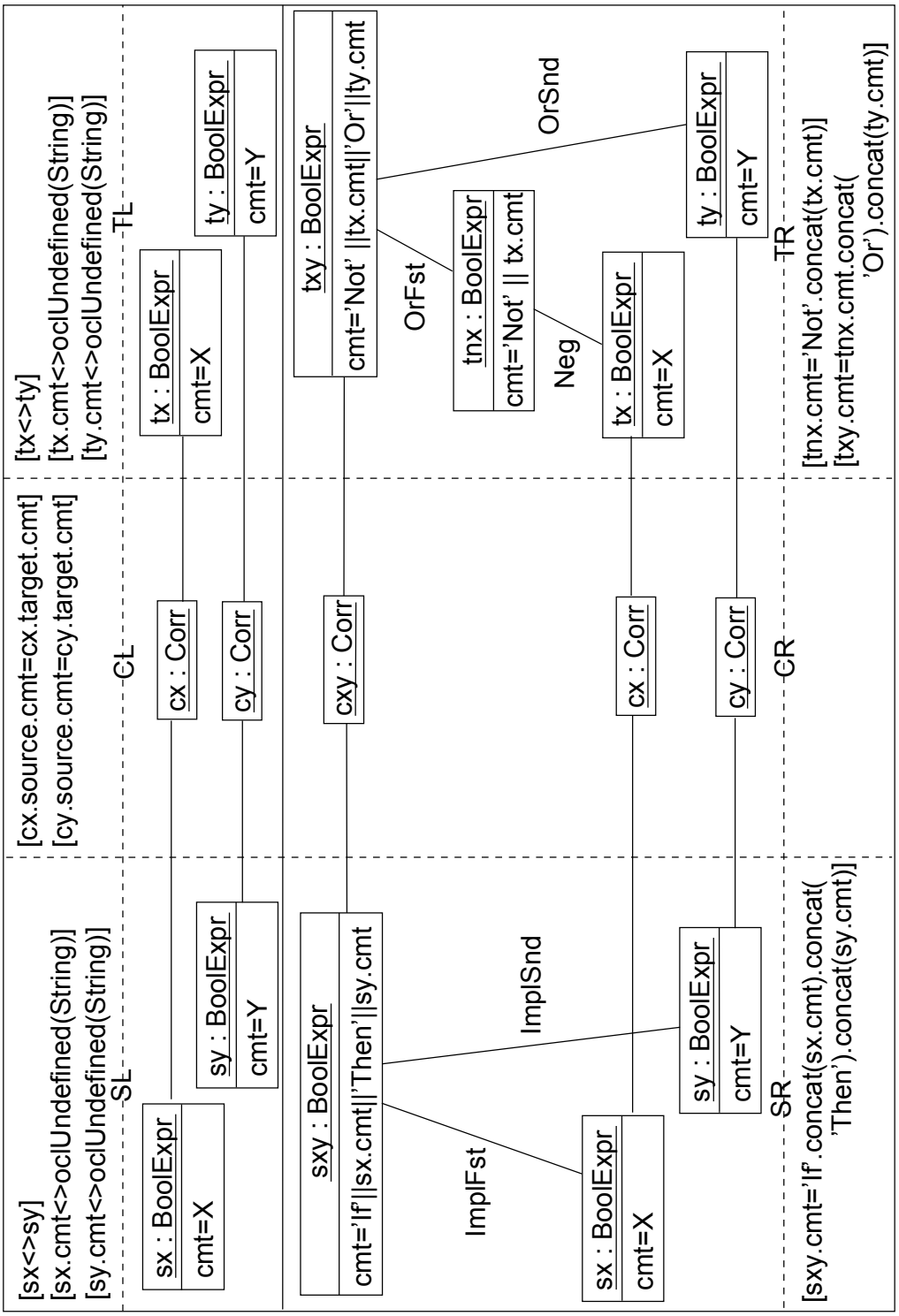


Figure 4.6: Example of OCL application conditions for triple rules.

Figure 4.6 depicts in the source part (SL,SR) and in the target part (TL,TR) OCL conditions as restrictions on the triple rule. OCL conditions in SL and TL are part of application preconditions, ensuring that the two input expressions X and Y do not coincide with each other. OCL conditions in SR and TR are part of application postconditions, specifying effects of the rule application: The two input expressions are unchanged and attribute values of new expressions are defined.

4.3.2 OCL Conditions for the Correspondence Part

OCL conditions in the correspondence part of a triple rule consist of OCL conditions for the correspondence between source and target parts and OCL conditions for the plain rule that corresponds to the correspondence part.

Figure 4.6 shows in (CL,CR) OCL conditions for the correspondence part of the example triple rule. OCL conditions in CL are part of application preconditions of the triple rule, restricting the correspondence between expressions in source and target models: Two cmt attributes must coincide with each other. OCL conditions in CR are part of application postconditions, stating that the correspondence between the input expressions is unchanged.

OCL conditions in the metamodel of the correspondence part need to be explicitly presented in the triple rule when the metamodel is automatically constructed from triple rules. For example, the correspondence part of the example triple rule in Fig. 4.4 includes the constraint for the meta class Cm2Dm: `self.classCM.name=self.classDM.name`

4.3.3 OCL Application Conditions of Triple Rules

OCL application conditions of a triple rule can be defined as a combination of OCL conditions in parts of the triple rule. For example, Fig. 4.6 depicts BACs of the example triple.

Definition 4.4. (OCL Application Conditions of Triple Rules)
OCL application conditions (BACs) of a triple rule consist of OCL conditions in source, target, and correspondence parts of the triple rule. BACs within the LHS and RHS of the triple rule are application pre- and postconditions, respectively. We have

- $BAC_{pre} = BAC_{SL} \cup BAC_{CL} \cup BAC_{TL}$,

- $BAC_{post} = BAC_{SR} \cup BAC_{CR} \cup BAC_{TR}$, and
- $BAC = [BAC_{pre}, BAC_{post}]$,

where BAC_{xx} with $xx \in \{`SL', `SR', `CL', `CR', `TL', `TR'\}$ are BACs in the LHS and RHS of the source, correspondence, and target parts of the triple rule, respectively; BAC_{pre} and BAC_{post} are application pre- and postconditions, respectively. □

Definition 4.5. (Application Condition Fulfillment of Triple Rules) A transformation triple rule with BACs is a tuple $tr = (L, R, BAC)$, where BAC includes OCL application conditions.

A triple graph H is derived from a triple graph G by a triple rule $tr = (L, R, BAC)$ and a triple match m iff:

- H is derived by $(L \rightarrow R, m)$ and
- BAC is fulfilled in the rule application $G \overset{r}{\Rightarrow} H$,

where $r : L \rightarrow R$ is a plain rule, which can be obtained from tr by viewing triple graphs as plain graphs. □

We can confirm that the triple rule shown in Fig. 4.6 is applicable for the transformation mentioned in Fig. 3.6.

4.4 OCL Conditions for Derived Triple Rules

Derived triple rules allow us to define operational scenarios of triple rules for model transformation as explained in Section 3.5. For a transformation from a source model to a target model, we assume the source and target models are parts of a triple graph that is produced by a TGG. In other words, we need to define such a triple graph for a model transformation. This triple graph can be defined using a triple derivation from the initial triple graph (S) to this triple graph. We can also use derived triple rules in a derivation that starts from the source model to define this triple graph. This is illustrated in Fig. 4.7.

For a transformation from H_S to H_T as shown in Fig. 4.7, we need to define the triple graph $H = (H_S \leftarrow H_C \rightarrow H_T)$ that can be produced by a TGG. In other words, we need to define a derivation from the initial triple graph

$(\phi \leftarrow \phi \rightarrow \phi)$ to H. In other way the resulting graph H can be obtained by the derivation $\{trF_k\}$ on the right-hand side as depicted in Fig. 4.7. This derivation is built by derived triple rules for forward transformation.

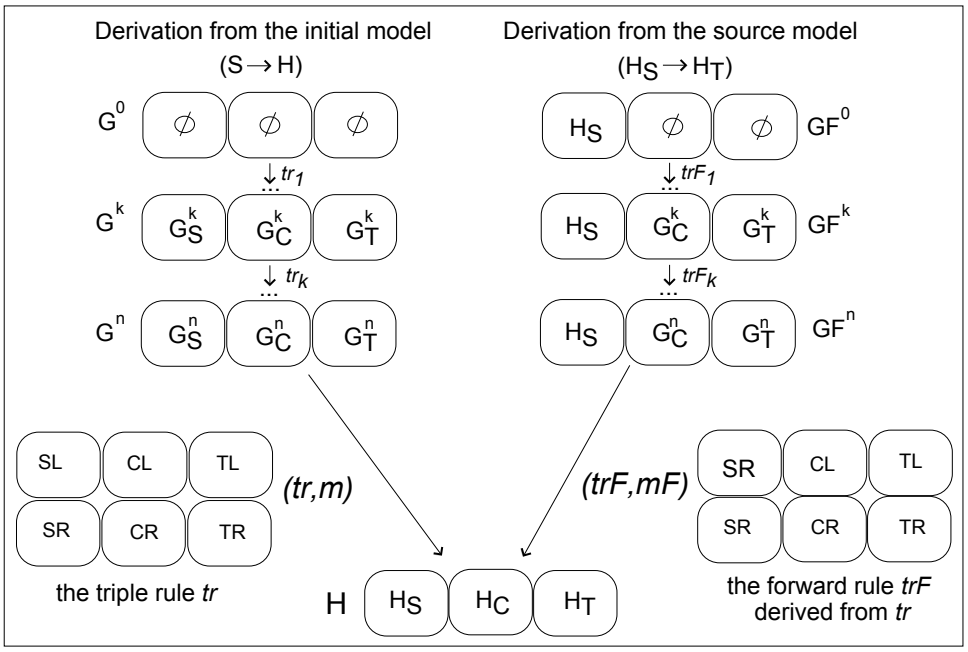


Figure 4.7: The correspondence between triple derivations of triple rules and derived triple rules.

A derivation with derived triple rules, e.g., $\{trF_k\}$ is always defined in relation to a corresponding derivation with the original triple rules, $\{tr_k\}$ in this case. Therefore, ACs of derived triple rules are also defined in such a context. Specially, whenever a triple rule tr is applicable in a triple match m, e.g., $G^n \overset{tr,m}{\Longrightarrow} H$ as shown in Fig. 4.7, a derived triple rule of this rule is also applicable in a corresponding triple match mF, i.e., $GF^n \overset{trF,mF}{\Longrightarrow} H$ as presented in Fig. 4.7. Based on the relationship between m and mF, e.g., as in Fig.4.7 (note that $G^i_S \subset H_S$ for $i = \overline{1,n}$), we can define ACs of derived rules as follows.

- BACs within a part of a derived triple rule (trF,trF, or trI) in which the LHS and the RHS do not coincide with each other are all BACs in the corresponding part of the triple rule tr.
- BACs (both pre- and postconditions) within a part of a derived triple rule (trF,trF, or trI) in which the LHS and the RHS coincide with each other are all BACs in the RHS of the corresponding part of the triple

rule tr, excepting OCL constraints with the keyword '@pre'. Note the keyword '@pre' is used to refer the state of items of the host graph before applying a rule. Hence, OCL constraints with '@pre' are inapplicable when the LHS and RHS of parts coincide with each other.

Formally, pre- and postconditions of derived triple rule can be defined as follows.

Definition 4.6. (Pre- and Postconditions of Derived Triple Rules) Let a triple rule tr be given. Preconditions of derived triple rules of tr are defined as follows.

- $BAC^{trF}_{pre} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CL} \cup BAC^{tr}_{TL}$,
- $BAC^{trB}_{pre} = BAC^{tr}_{SL} \cup BAC^{tr}_{CL} \cup BAC^{tr}_{TR*}$, and
- $BAC^{trI}_{pre} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CL} \cup BAC^{tr}_{TR*}$

Postconditions of derived triple rules of tr are defined as follows.

- $BAC^{trF}_{post} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CR} \cup BAC^{tr}_{TR}$,
- $BAC^{trB}_{post} = BAC^{tr}_{SR} \cup BAC^{tr}_{CR} \cup BAC^{tr}_{TR*}$, and
- $BAC^{trI}_{post} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CR} \cup BAC^{tr}_{TR*}$,

where

- BAC^{trF}_{pre}, BAC^{trB}_{pre}, and BAC^{trI}_{pre} are the precondition of derived rules for forward transformation, backward transformation, and model integration, respectively; BAC^{trF}_{post}, BAC^{trB}_{post}, and BAC^{trI}_{post} are the postcondition of the derived rules,
- BAC^{tr}_{xx} with $xx \in \{`SL', `SR', `CL', `CR', `TL', `TR'\}$ are BACs in the LHS and RHS of parts of the triple rule tr, respectively, and
- BAC^{tr}_{SR*}, BAC^{tr}_{CR*}, and BAC^{tr}_{TR*} are BACs in SR, CR, and TR excepting OCL constraints with '@pre', respectively. □

For example, the precondition of the forward rule derived from the triple rule shown in Fig. 4.6 include OCL conditions in `SR`, `CL`, and `TL`. The postcondition of this triple rule include OCL conditions in `SR`, `CR`, and `TR`. All OCL conditions in the example triple rule do not contain '@pre'.

4.5 Presenting TGGs Incorporating OCL

We propose the USE4TGG language in order to present the incorporation of OCL in TGGs as well as to realize this integration within a UML-based metamodeling framework like USE [GBR07].

4.5.1 Patterns of TGGs Incorporating OCL

As the result from the analysis of patterns in QVT and TGGs, as well as patterns mentioned in [BG06], we propose a hybrid form of patterns for TGGs incorporating OCL as shown in Fig. 4.8.

```
TGG&OCL Pattern = { object: Class_i
                    (object1, object2): Association_j
                    [OCL_Condition_k]}
```

Figure 4.8: The form of patterns for TGGs incorporating OCL.

Like QVT, our form of patterns allows us to add OCL constraints in a flexible way. Objects and links in patterns can be seen as object diagrams so that OCL constraints can be employed to restrict these objects and links in the patterns.

A substantial part of our form of patterns has a one-one mapping to TGG patterns because both share similar concepts, namely objects and links. In QVT, one has to employ assignments or OCL expressions in order to specify links. Our form of patterns can keep the TGG spirit of a declarative specification since one is not forced to write assignments, which indicate operational specifications.

Our form of patterns allows us to add OCL constraints (invariants) for arbitrary restrictions on metamodels. QVT separates the constraints from rule specifications. But in some cases invariants need to be explicitly seen as pre- or postconditions of rule applications.

4.5.2 Descriptions in USE4TGG

Figure 4.9 depicts a textual concrete syntax of the language USE4TGG, and Fig. 4.10 presents the triple rule `addAssociation` incorporating OCL in the language USE4TGG. This triple rule is also shown in Fig. 4.4.

```
rule <ruleName>
[mode NONDELETING|EXPLICT]
checkSource(
  <TGGs&OCL Pattern>
){
  <TGGs&OCL Pattern>}
checkTarget(
  <TGGs&OCL Pattern>
){
  <TGGs&OCL Pattern>}
checkCorr(
  <Ext TGGs&OCL Pattern>
){
  <Ext TGGs&OCL Pattern>}
end
```

Figure 4.9: Concrete syntax of the language USE4TGG.

A triple rule in USE4TGG includes three groups which are recognized by the keywords 'checkSource', 'checkTarget', and 'checkCorr'. They correspond to the source, target, and correspondence parts of the triple rule. Each group includes two parts that correspond to the LHS and RHS of parts of the rule. The first part is enclosed in parentheses '(' and ')'. The second part is grouped by braces '{' and '}'.

USE4TGG has two modes for specifying triple rules incorporating OCL. The first mode is indicated with the keyword NONDELETING, and the second mode is indicated with the keyword EXPLICIT. The first mode is used for specifying non-deleting rules, e.g., for treating the example transformation in this chapter. This mode is the default and the keyword NONDELETING may be skipped. The second mode is used for specifying rules which may be deleting. In the second case, all nodes and links must be explicitly stated whereas in the first case only the added nodes and links are mentioned. The pattern format of the two modes is identical. For specifying non-deleting rules, the first mode is much shorter than the second mode.

The language USE4TGG offers a particular feature to specify correspondences between source and target parts of a triple rule. The syntax for these so-called correspondence links is as follows.

(object_S, object_T) as (role_S, role_T) in object_C: Class_C

The items S, T, and C stand for Source, Target and Correspondence, respectively. The advantage of this extension is that we can automatically generate the metamodel of the correspondence domain. For example, the correspondence links in the 'checkCorr' part as presented in Fig. 4.10 allows us to

```
rule addAssociation
checkSource(
  oneCM:ClassCM
  manyCM:ClassCM
){
  assoc:Assoc
  oneRole:AssocEnd
  manyRole:AssocEnd
  (assoc,oneRole):Assoc_AssocEnd
  (assoc,manyRole):Assoc_AssocEnd
  (oneCM,oneRole):Class_AssocEnd
  (manyCM,manyRole):Class_AssocEnd
  [oneRole.multiplicity='1']
  [oneRole.roleName<>oclUndefined(String)]
  [manyRole.multiplicity= '*']
  [manyRole.roleName<>oclUndefined(String)]}
checkTarget(
  oneDM:ClassDM
  manyDM:ClassDM
){
  oneAttr:Attr
  manyAttr:Attr
  oneOp:Op
  manyOp:Op
  (oneDM,manyAttr):Class_Attr
  (manyDM,oneAttr):Class_Attr
  (oneDM,manyOp):Class_Op
  (manyDM,oneOp):Class_Op
  [oneAttr.type=oneDM.name]
  [oneOp.retType=oneDM.name]
  [manyAttr.type='Set('.concat(manyDM.name).concat(')')]
  [manyOp.retType=manyAttr.type]}
checkCorr(
  (oneCM,oneDM) as (classCM,classDM) in oneMapping:Cm2Dm
  (manyCM,manyDM) in manyMapping:Cm2Dm
  Cm2Dm:[self.classCM.name=self.classDM.name]
){
  (oneRole,oneAttr) as (roleCls,attr) in one2Attr:AssocEnd2Attr
  (oneRole,oneOp) as (roleCls,op) in one2Attr:AssocEnd2Attr
  (manyRole,manyAttr) in many2Attr:AssocEnd2Attr
  (manyRole,manyOp) in many2Attr:AssocEnd2Attr
  AssocEnd2Attr:[self.attr.name=self.roleCls.roleName]
  AssocEnd2Attr:[self.op.name='get'.concat(self.roleCls.roleName)]}
end
```

Figure 4.10: The triple rule `addAssociation` in USE4TGG.

derive the metamodel of the correspondence domain as presented in Fig. 4.2. In addition, the links allow us to suppress links connecting the source and target models, i.e., the links (object_S, object_C) and (object_T, object_C) do not have to be given explicitly. They can be inferred from the correspondence link. Correspondence links highlight the correspondence in a condensed way.

OCL conditions from the correspondence part can become invariants on the metamodel. For example, OCL conditions in the 'checkCorr' part shown in Fig. 4.10 can be employed as invariants for the metamodel shown in Fig. 4.2.

4.6 Related Work

Triple Graph Grammars (TGGs) have been proposed first in [Sch95]. Since then, a lot of further developments and applications have indicated the strength of TGGs in particular within software engineering.

In [GK07], a comparison between TGGs and the QVT Core language is presented. The paper shows that both approaches are very similar. A difference between them is the use of OCL constraints within the QVT Core language whereas multiple correspondence nodes would do the same within TGG. That paper rises the question how to integrate OCL into TGGs. This can be seen as one motivation for our work.

In [KW07], a quite complete survey of principles, concepts, usability, and implementations of TGGs is discussed. The report proposes a notation for OCL constraints within TGG descriptions. However, OCL constraints within that approach are only for attribute expressions. The issues OCL constraints as application conditions of rules and OCL constraints on metamodels are not mentioned in that work.

The approach in [SZG06] considers the integration of OCL into model transformations within Fujaba. This is realized by integrating the Dresden OCL Toolkit. However, the paper does not mention explicitly TGG concepts as well as OCL constraints in connection with TGGs.

The work in [KS04] proposes an approach for the integration of a number of MOF-compliant models. So-called Multi-Domain Integration (MDI) rules, a generalization of TGGs, are proposed. The paper mentions the value of declarative specifications in connection with TGGs and OCL. However, the specification of OCL constraints within TGGs is not treated.

In [KS06, Kö05, GGL05], an extension of TGGs for model integration is proposed. The basic idea is from declarative TGG rules one can derive

operational rules for consistency checking, consistency recovery and model transformation. However, within these works, OCL constraints, which are in our view central for such approaches, are not treated.

The work in [HKT02a] outlines an approach utilizing TGGs for managing consistency between views in general and UML models and semantic domains in particular. In [GdL06b], TGGs are employed for model view management. The work in [GW06] presents a TGG approach for incremental model synchronization. Our work adapts from these papers the intended application scenarios.

Plain textual OCL has been extended to be visualized with collaborations in [BKPPT01]. The work in [ZHG05] proposes using OCL conditions as application preconditions of plain rules. Our work extends this approach for treating TGGs.

Summary

In this chapter we have proposed an approach for the integration of TGGs and OCL. We take QVT as the key motivation. We aim to promote the integration of TGGs in model-driven approaches, especially in the context of QVT. The incorporation of OCL in TGGs offers a declarative description of transformation relationships and the correspondence between source and target models, which can be compared to the QVT Relational language.

We have formally explained OCL application conditions of triple rules and derived triple rules. Using OCL application conditions is an approach for increasing the expressiveness of transformation rules, especially in the context of model-driven approaches.

The language USE4TGG has been proposed in this chapter for presenting the integration of TGGs and OCL. This language is the key for realizing this integration within UML-based metamodeling frameworks.

This chapter has explained the integration of TGGs and OCL as the declarative description for model transformation. In the next chapter we focus on a corresponding operational description for model transformation.

Chapter 5

Operationalizing TGGs Incorporating OCL

This chapter focuses on the operational description of model transformations based on triple rules incorporating OCL. We employ OCL in order to realize the operational scenarios of triple rules towards an OCL-based framework for model transformation. In this framework new operations derived from triple rules for model synchronization are proposed. We translate triple rules incorporating OCL into operations in OCL. These operations are realized by taking two views on them: Declarative OCL pre- and postconditions are employed as operation contracts, and command sequences are taken as an operational realization. Our approach is realized in the tool UML-based Specification Environment (USE). This approach not only covers a complete realization of TGGs incorporating OCL in USE, but also offers means for quality assurance of model transformation.

5.1 Introduction

Chapter 4 shows that the incorporation of OCL in TGGs offers a declarative representation of model transformation. This allows us to specify what a model transformation does. It is necessary to explain how the model transformation is accomplished.

Current works often represent triple graphs as plain graphs for an implementation of TGGs: The three graphs are modeled as a single integrated graph, where the embedding of the correspondence graph is represented by additional edges. Their aim is to employ tools developed for typed attributed graph transformation such as Fujaba [KRW04, KW07], AToM3 [GdL06b], and AGG [dLBE+07] in order to implement model transformations based on TGGs. However, some problems arise from this approach. First, the trace between a triple rule and derived rules as well as the trace between a derived triple rule and its realization are not kept. This prevents us from tracing be-

tween the declarative and operational descriptions of model transformations, whereas this trace is necessary for verification and validation of transformations. Second, the operational description of transformations in this way is often difficult for users to access and check it since it is presented at the low level of abstraction (such as the Java level). Third, restrictions on triple rules and derived triple rules are not mentioned, and the well-formedness constraints of models are ignored within these works. These restrictions are also necessary for verifying and validating transformations.

We propose an approach based on OCL for the operational description of TGGs incorporating OCL. A triple rule is mapped to an OCL operation. The operation is realized by taking two views on it: Declarative OCL pre- and postconditions are employed as operation contracts, and command sequences are taken as an operational realization. The main benefits of this approach include:

- Operation preconditions allow us to check if a (derived) triple rule is applicable. After each rule application, we can check the postconditions of the rule for an on-the-fly verification of the transformation.
- We can obtain the operational description of TGGs incorporating OCL by generating it from the corresponding declarative description in USE4TGG. This translation allows us to keep the trace between these descriptions.
- This OCL-based approach allows us to propose new operations derived from triple rules for model synchronization.

We implement TGGs incorporating OCL in USE [GBR07] towards an OCL-based framework for model transformation. While the declarative description in USE4TGG is similar to the QVT Relational language, this OCL-based framework can be compared to the QVT Operational Mapping language. This OCL-based framework offers means for quality assurance of model transformation such as ensuring well-formed models, checking properties of models, verifying transformations, and maintaining model consistency.

The rest of this chapter is organized as follows. Section 5.2 presents the translation of triple rules incorporating OCL into an OCL-based representation. Section 5.3 focuses on operational scenarios of triple rules incorporating OCL for model transformation. Section 5.4 presents quality assurance of model transformation. Section 5.5 discusses on related work. This chapter is closed with a summary.

5.2 Translating Triple Rules to OCL

This section explains how we can translate triple rules incorporating OCL into OCL operations. The basic idea of this translation originates from the mapping between a graph transformation system and an object-oriented system as mentioned in Chapter 3 and in [GBD08, BG06, ZHG05]: (1) Host graphs correspond to system states as object diagrams, (2) a graph transformation rule corresponds to a collaboration as a context in which the implementation of an operation executes, and (3) the LHS and RHS of a rule within a match correspond to system snapshots (system states in form of object diagrams) as the pre- and postconditions of the corresponding collaboration. For example, the rule application shown in Fig. 3.3 can be seen as the execution of an operation, transiting the system snapshot (corresponding to the LHS) to the next system snapshot (corresponding the RHS). This execution can also be described as a sequence of state transitions. Each transition is executed by a basic state manipulation. These manipulations include creating and destroying objects (nodes) and links (edges) and modifying attributes.

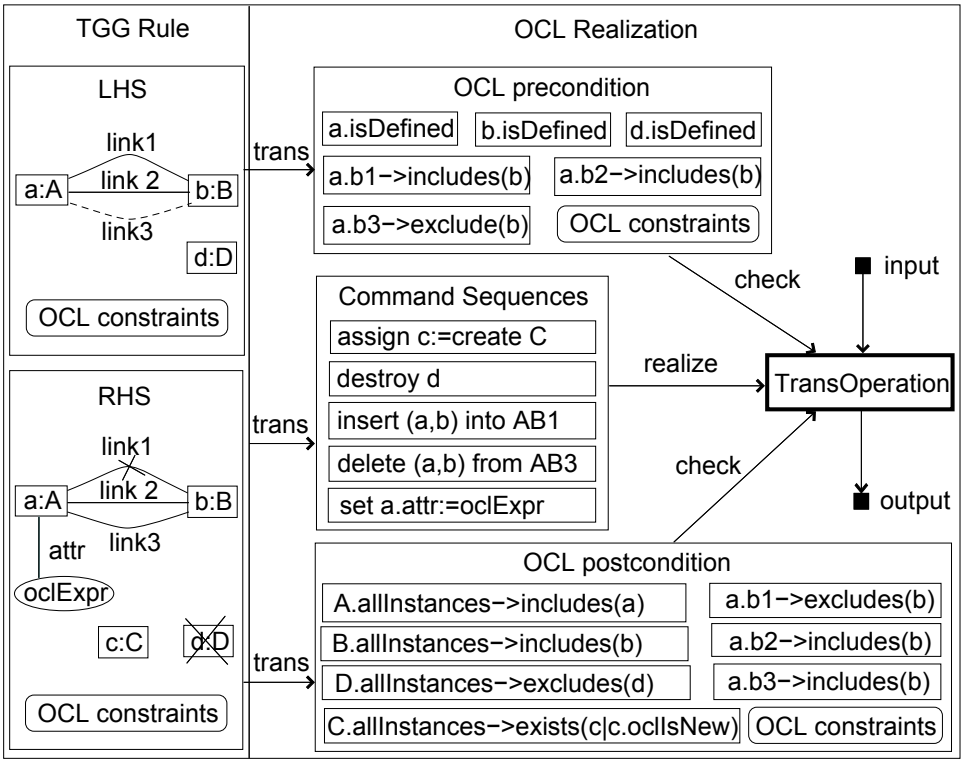


Figure 5.1: OCL realization for a TGG rule: (1) By declarative OCL pre- and postconditions, and (2) by command sequences

Triple rules can also be realized in such a way since we can consider them as plain rules, according to Subsect. 3.6.1. This can be illustrated as in Fig. 5.1. In the description of a triple rule we need to keep separate parts (objects, links, and conditions) that corresponds to (source, target, and correspondence) parts of the triple rule. This separation allows us to trace between the operational and declarative descriptions of triple rules. It also makes the operational description more understandable for users.

5.2.1 Matching Triple Rules Incorporating OCL

When a rule is applied, the LHS is matched to the host graph such that nodes and edges of the LHS match to objects and links of the host graph, and OCL conditions in the LHS are fulfilled by the match. Such a matching can be seen as a query on objects. The query condition is formed by OCL conditions in the LHS and constraints of links. Therefore, we can map a rule to an operation as follows: The LHS corresponds to the input, and the query condition is the precondition of the operation.

```
context RuleCollection::tgg4bool_coEvol(
      matchSL:Tuple(sx:BoolExpr,sy:BoolExpr),
      matchTL:Tuple(tx:BoolExpr,ty:BoolExpr),
      matchCL:Tuple(cx:Corr,cy:Corr))
pre tgg4bool_coEvol_pre:
--matchSL:Tuple(sx:BoolExpr,sy:BoolExpr)
  let sx:BoolExpr = matchSL.sx in
  let sy:BoolExpr = matchSL.sy in
--matchTL:Tuple(tx:BoolExpr,ty:BoolExpr)
  let tx:BoolExpr = matchTL.tx in
  let ty:BoolExpr = matchTL.ty in
--matchCL:Tuple(cx:Corr,cy:Corr)
  let cx:Corr = matchCL.cx in
  let cy:Corr = matchCL.cy in
  --S_precondition
    sx<>sy and
    sx.cmt<>oclUndefined(String) and
    sy.cmt<>oclUndefined(String) and
  --T_precondition
    tx<>ty and
    tx.cmt<>oclUndefined(String) and
    ty.cmt<> oclUndefined(String) and
  --C_precondition
    Set{cx.upper}->includesAll(Set{sx}) and
    Set{cx.lower}->includesAll(Set{tx}) and
    Set{cy.upper}->includesAll(Set{sy}) and
    Set{cy.lower}->includesAll(Set{ty})
```

Figure 5.2: OCL precondition for the triple rule shown in Fig. 4.6

The precondition of transformation operations can be represented in OCL

as illustrated in Fig. 5.1. For each object `a` in the LHS, the query condition must include an `a.isDefined` condition stating the existence of the object `a`. For each edge `link1` in the LHS linking two objects `a` (of type `A`) and `b` (of type `B`) in LHS, the query condition must include an `a.b1→includes(b)` condition. This condition means that `b` must be included in the set of objects retrieved when navigating from `a` to the related objects in `B`. The association end to use in the navigation `b1` is extracted from the metamodel (as a class diagram) according to the type of the two object participants and the type (association) of the link `link1`. OCL conditions in the LHS map to a conjunction of conditions within the query condition. When the multiplicity at the association end `b1` is 0..1 or 1..1, the OCL condition becomes `a.b1=b`, or we can use `a.b1→includesAll(Set{b})`.

For each pair of objects in the LHS which are newly linked in the RHS, e.g., `link3` in Fig. 5.1, the query condition must include an `a.b3→exclude(b)` condition, that states the object `a` is not linked to the object `b` by any link with the same type as `link3`.

The input of a transformation operation is represented by parameters `matchSL`, `matchTL`, and `matchCL` of the Tuple type in OCL. They correspond to the patterns `SL`, `CL`, and `TL` of the corresponding triple rule. Figure 5.2 depicts the precondition of the operation for the triple rule shown in Fig. 4.6. We organize transformation operations as methods of a common class `RuleCollection`.

5.2.2 Checking the Postcondition of Rule Applications

Effects of a rule application include (1) the preservation, creation, and deletion of nodes (objects) and edges (links), and (2) the update of attribute values. Figure 5.1 illustrates effects of a rule application: (1) The link `link1` is deleted, the link `link2` is preserved, and the link `link3` is created; (2) the objects `a` and `b` are preserved, the object `c` is newly created, and the object `d` is deleted; (3) the attribute of the object a is updated. The effects can be expressed by OCL conditions as shown in Fig. 5.1. We can obtain the operation postcondition of the rule as the combination of these OCL conditions.

A link condition in the RHS of a rule, e.g., `a.c1→includes(c)` may refer to a new object, the object `c` in the case. Then, for the operation postcondition of the rule, the link condition must be included in the OCL condition that declares the new object, i.e., we have `C.allInstances→exists(c|c.oclIsNew and a.c1→includes(c))`.

```
context RuleCollection::tgg4bool_coEvol(
      matchSL:Tuple(sx:BoolExpr,sy:BoolExpr),
      matchTL:Tuple(tx:BoolExpr,ty:BoolExpr),
      matchCL:Tuple(cx:Corr,cy:Corr))
post tgg4bool_coEvol_post:
--matchSL:Tuple(sx:BoolExpr,sy:BoolExpr)
 let sx: BoolExpr = matchSL.sx in
 let sy: BoolExpr = matchSL.sy in
--matchTL:Tuple(tx:BoolExpr,ty:BoolExpr)
 let tx: BoolExpr = matchTL.tx in
 let ty: BoolExpr = matchTL.ty in
--matchCL:Tuple(cx:Corr,cy:Corr)
 let cx: Corr = matchCL.cx in
 let cy: Corr = matchCL.cy in
 --S_postcondition
  BoolExpr.allInstances->includesAll(Set{sx,sy}) and
  BoolExpr.allInstances->exists( sxy | sxy.oclIsNew and
   Set{sxy.implFst}->includesAll(Set{sx}) and
   Set{sxy.implSnd}->includesAll(Set{sy}) and
   sxy.cmt='If'.concat(sx.cmt).concat(
                                  'Then').concat(sy.cmt)and
  --T_postcondition
   BoolExpr.allInstances->includesAll(Set{tx,ty}) and
   BoolExpr.allInstances->exists( txy | txy.oclIsNew and
    BoolExpr.allInstances->exists( tnx | tnx.oclIsNew and
     Set{txy,tnx}->size = 2 and
     Set{txy.orFst}->includesAll(Set{tnx}) and
     Set{txy.orSnd}->includesAll(Set{ty}) and
     Set{tnx.neg}->includesAll(Set{tx}) and
     tnx.cmt='Not'.concat(tx.cmt) and
     txy.cmt=tnx.cmt.concat('Or').concat(ty.cmt) and
    --C_postcondition
      Corr.allInstances->includesAll(Set{cx,cy}) and
      Corr.allInstances->exists( cxy | cxy.oclIsNew and
       Set{cxy.upper}->includesAll(Set{sxy}) and
       Set{cxy.lower}->includesAll(Set{txy}) and
       Set{cx.upper}->includesAll(Set{sx}) and
       Set{cx.lower}->includesAll(Set{tx}) and
       Set{cy.upper}->includesAll(Set{sy}) and
       Set{cy.lower}->includesAll(Set{ty})))))
```

Figure 5.3: OCL postcondition for the triple rule shown in Fig. 4.6

We organize the operation postcondition of a triple rule in parts corresponding to the patterns (SL, CL, and TL) of the triple rule as depicted in Fig. 5.3. There is a restriction on the order of OCL conditions. A new link in the correspondence part of a triple rule may refer to new objects of target and source parts. In this case the `exists` condition declaring new objects must include this link condition. Therefore, for the OCL postcondition of the triple rule, OCL conditions of the correspondence part (CL) must be included in OCL conditions of target and source parts. The order of OCL conditions in the source part (SL) is independent with the one in the target part (TL).

5.2.3 Rewriting Triple Graphs

We describe the execution of a transformation operation as a sequence of state transitions, where each transition is executed by a basic state manipulation. These manipulations include (1) creating and destroying objects (nodes) and links (edges) and (2) modifying attributes.

```
!openter rc tgg4bool_coEvol(matchSL,matchTL,matchCL)
--matchSL:Tuple(sx:BoolExpr,sy:BoolExpr)
!let _sx = matchSL.sx
!let _sy = matchSL.sy
--matchTL:Tuple(tx:BoolExpr,ty:BoolExpr)
!let _tx = matchTL.tx
!let _ty = matchTL.ty
--matchCL:Tuple(cx:Corr,cy:Corr)
!let _cx = matchCL.cx
!let _cy = matchCL.cy
-------------------------Create for matchS
!assign _sxy := create BoolExpr
!insert(_sxy,_sx) into ImplFst
!insert(_sxy,_sy) into ImplSnd
!set _sxy.cmt := 'If'.concat(_sx.cmt).concat(
'Then').concat(_sy.cmt)
-------------------------Create for matchT
!assign _txy := create BoolExpr
!assign _tnx := create BoolExpr
!insert(_tnx,_tx) into Neg
!insert(_txy,_tnx) into OrFst
!insert(_txy,_ty) into OrSnd
!set _tnx.cmt := 'Not'.concat(_tx.cmt)
!set _txy.cmt:=_tnx.cmt.concat('Or').concat(_ty.cmt)
-------------------------Create for matchC
!assign _cxy := create Corr
!insert(_cxy,_sxy) into L_BoolExpr_Corr
!insert(_cxy,_txy) into R_BoolExpr_Corr
-------------------------------------------------
!opexit
```

Figure 5.4: USE command sequence for the triple rule shown in Fig. 4.6

Figure 5.1 shows in the part 'command sequences' the basic commands in USE syntax. There are restrictions of the order of the command sequence for a triple rule. First, because of the dependence relationship between links and objects in a rule application, e.g., an object may need to be created before the creation of a link, a command sequence for realizing the rule needs to be in the order: the creation of objects → the creation of links → the deletion of links → the deletion of objects. Second, we organize the command sequence in parts that correspond to parts of the triple rule. Finally, objects in the source part (`SR`) and ones in the target part (`TR`) are not linked to each other, and they may be linked to objects in the correspondence part (`CR`). Therefore, we can organize the command sequence in the order: the creation (of objects and links) in `SR` → the creation in `TR` → the creation in `CR` → the deletion in `CR` → the deletion in `SR` → the deletion in `TR`. Figure 5.4 presents the USE command sequence for the triple rule shown in Fig 4.6.

A part of OCL conditions in the RHS in an assignment-like style can be translated into '`set`' commands in USE which manipulate attribute values. Although we do not force to use assignments in the RHS, certain OCL formulas must be presented in the assignment-like style, if we want to obtain a complete operation realization of a triple rule in form of a command sequence: Only if in the bottom part of a triple rule an attribute modification (written down as an OCL formula) is presented in the form `Attribute` =`OclExpression`, the resulting operation is realized in a correct way.

5.2.4 Implementation in USE4TGG

Figure 5.5 depicts the description in USE4TGG of the triple rule shown in Fig. 4.6. We can translate such a description into a declarative representation of the triple rule as shown in Fig. 5.2 and Fig. 5.3, and the operational representation of the triple rule as shown in Fig. 5.4.

The feature of USE4TGG about generating invariants in the metamodel of correspondence parts of triple rules has been introduced in Chapter 4. In the next section we see the other features of USE4TGG: The USE4TGG description of triple rules incorporating OCL can be translated into operations for forward and backward transformations, model integration, and model synchronization.

```
rule tgg4bool
checkSource{
  sx:BoolExpr
  sy:BoolExpr
  [sx<>sy]
  [sx.cmt<>oclUndefined(String)]
  [sy.cmt<>oclUndefined(String)]
}(
  sxy:BoolExpr
  (sxy,sx):ImplFst
  (sxy,sy):ImplSnd
  [sxy.cmt='If'.concat(sx.cmt).concat(
                        'Then').concat(sy.cmt)])
checkTarget{
  tx:BoolExpr
  ty:BoolExpr
  [tx<>ty]
  [tx.cmt<>oclUndefined(String)]
  [ty.cmt<> oclUndefined(String)]
}(
  txy:BoolExpr
  tnx:BoolExpr
  (tnx,tx):Neg
  (txy,tnx):OrFst
  (txy,ty):OrSnd
  [tnx.cmt='Not'.concat(tx.cmt)]
  [txy.cmt=tnx.cmt.concat('Or').concat(ty.cmt)])
checkCorr{
  (sx,tx) as (upper,lower) in cx:Corr
  (sy,ty) in cy:Corr
}(
  (sxy,txy) in cxy:Corr)
end
```

Figure 5.5: USE4TGG description for the triple rule shown in Fig. 4.6

5.3 Model Transformation based on TGGs and OCL

According to Chapter 4, the declarative description in USE4TGG of a triple rule incorporating OCL can be translated into the derived triple rules and their OCL application conditions. This section explains how the derived triple rules as the result of this translation can be presented in OCL. This OCL realization of TGGs incorporating OCL offers an OCL-based framework for model transformation. We illustrate this point by means of examples.

5.3.1 Overview of Operations for Derived Triple Rules

A triple rule derived from a triple rule can be obtained by reorganizing patterns (SL,SR,CL,CR,TL, and TR) of the original triple rule as shown in Definition 3.17. The input of operations for the derived triple rules can be defined as illustrated in Fig. 5.6.

Operational Scenarios	Input/Computing
Model Co−Evolution	I I I ? ? ?
Forward Transformation	I I I I ? ?
Model Integration	I I I I ? I
Model Synchronization	I I I U I U

I: Input
?: Create
U: Update

SL	CL	TL
SR	CR	TR

maskS=SR\SL
maskC=CR\CL
maskT=TR\TL

Figure 5.6: Input and computation for derived triple rules

We use a sheet including six cells that correspond to six patterns of the original triple rule in order to describe the input of each operation. The cell denoted by 'I' means that nodes in this part belong to the input of the operation. The cell denoted by '?' represents objects created by the operation. The cell denoted by 'U' means that part of this cell belongs to the input of the operation, and this part can be updated by the operation. The remaining nodes in this cell correspond to objects created by the operation.

Figure 5.7 presents operations for triple rules derived from the triple rule presented in Fig. 4.6. The USE4TGG description of this rule is shown in Fig. 5.5. From the implementation point of view, operations for forward transformations coincide with operators for backward transformations.

```
RuleCollection
-------------------------------------Model Co-evolution
tgg4bool_coEvol(matchSL:Tuple(sx:BoolExpr,sy:BoolExpr),
  matchTL:Tuple(tx:BoolExpr,ty:BoolExpr),
  matchCL:Tuple(cx:Corr,cy:Corr))
-------------------------------------Forward Transformation
tgg4bool_forwTrafo(
  matchSR:Tuple(sxy:BoolExpr,sx:BoolExpr,sy:BoolExpr),
  matchTL:Tuple(tx:BoolExpr,ty:BoolExpr),
  matchCL:Tuple(cx:Corr,cy:Corr))
-------------------------------------Model Integration
tgg4bool_integraTrafo(
  matchSR:Tuple(sxy:BoolExpr,sx:BoolExpr,sy:BoolExpr),
  matchTR:Tuple(sxy:BoolExpr,tx:BoolExpr,ty:BoolExpr),
  matchCL:Tuple(cx:Corr,cy:Corr)
-------------------------------------Model Synchronization
tgg4bool_synchTrafo(matchSL:Tuple(sx:BoolExpr,sy:BoolExpr),
  matchTL:Tuple(tx:BoolExpr,ty:BoolExpr),
  matchCL:Tuple(cx:Corr,cy:Corr), maskS:Tuple(sxy:BoolExpr),
  maskT:Tuple(txy:BoolExpr),maskC:Tuple(cxy:Corr))
```

Figure 5.7: Operations for derived triple rules from the triple rule `tgg4bool` shown in Fig. 4.6

5.3.2 Model Co-Evolution

A model co-evolution based on triple rules is a sequence of triple transformations for an integration model of the source and target models and a correspondence between them.

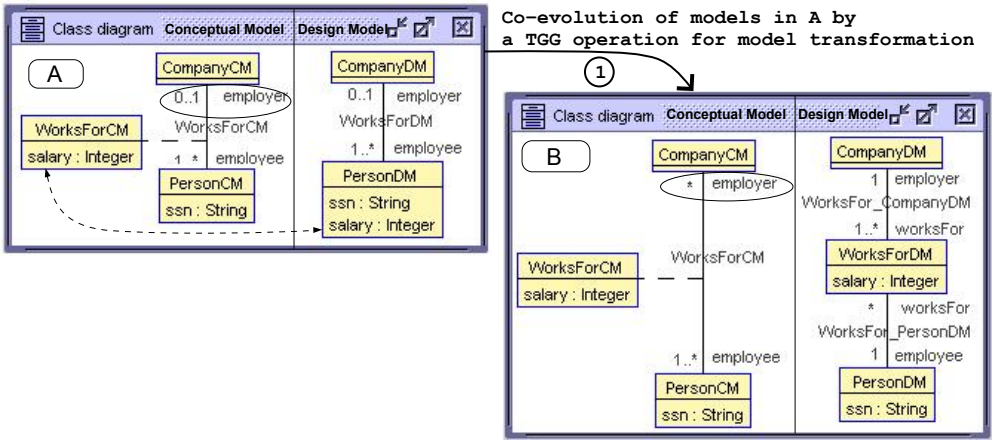


Figure 5.8: Model co-evolution by triple rules incorporating OCL

Figure 5.8 presents an example of model co-evolution. We consider a typical situation in software development, where one wants to formally trace the relationship between a conceptual model and a design model while both

models are evolving. In exemplary form, Fig. 5.8 illustrates the situation with model versions `A` and `B`. The conceptual (left) and design (right) models are presented in concrete syntax. But, the following discussion refers to the models as instances of the UML metamodel.

A co-evolution step between two model versions can be represented by a triple rule incorporating OCL. The LHS and RHS of the triple rule correspond to the target and source versions. The consistency between the conceptual and design models of a version is maintained by links and OCL constraints in the correspondence part of the triple rule. The co-evolution is carried out by a transformation operation that is derived from the triple rule: the source version is transformed to the target version. The note (1) in Fig. 5.8 shows a co-evolution from the version `A` to the version `B` by the triple rule `transAssocclass_one2many` when the association multiplicity in the conceptual model in the version `A` is changed. The correspondence between the two models in the versions `A` or `B` expresses constraints for a refinement of an association class in the conceptual model, e.g., names of corresponding `Class` (`Attribute`) objects must coincide. In the version `A`, we recognize the correspondence between two `Attribute` objects in which the `name` attribute is 'salary'. This correspondence (the dashed line in `A`) is represented by links that connect these `Attribute` objects with an object `Cm2Dm` in the correspondence part of the triple rule, and by the OCL invariant in the class `Cm2Dm`: `self.attrCM.name=self.attrDM.name`.

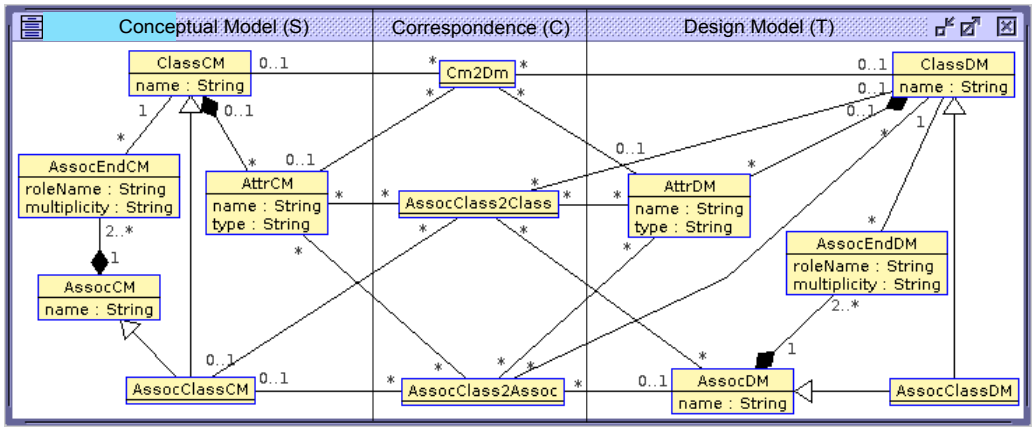


Figure 5.9: Simplified metamodel as a schema of triple rules for an integrated model of the conceptual and design models

Figure 5.9 shows the simplified metamodel as a schema of triple rules for an integrated model of the conceptual and design models. Note that the metamodels of the source and target models coincide with the UML metamodel. However, for an integration of source and target models we distinguish be-

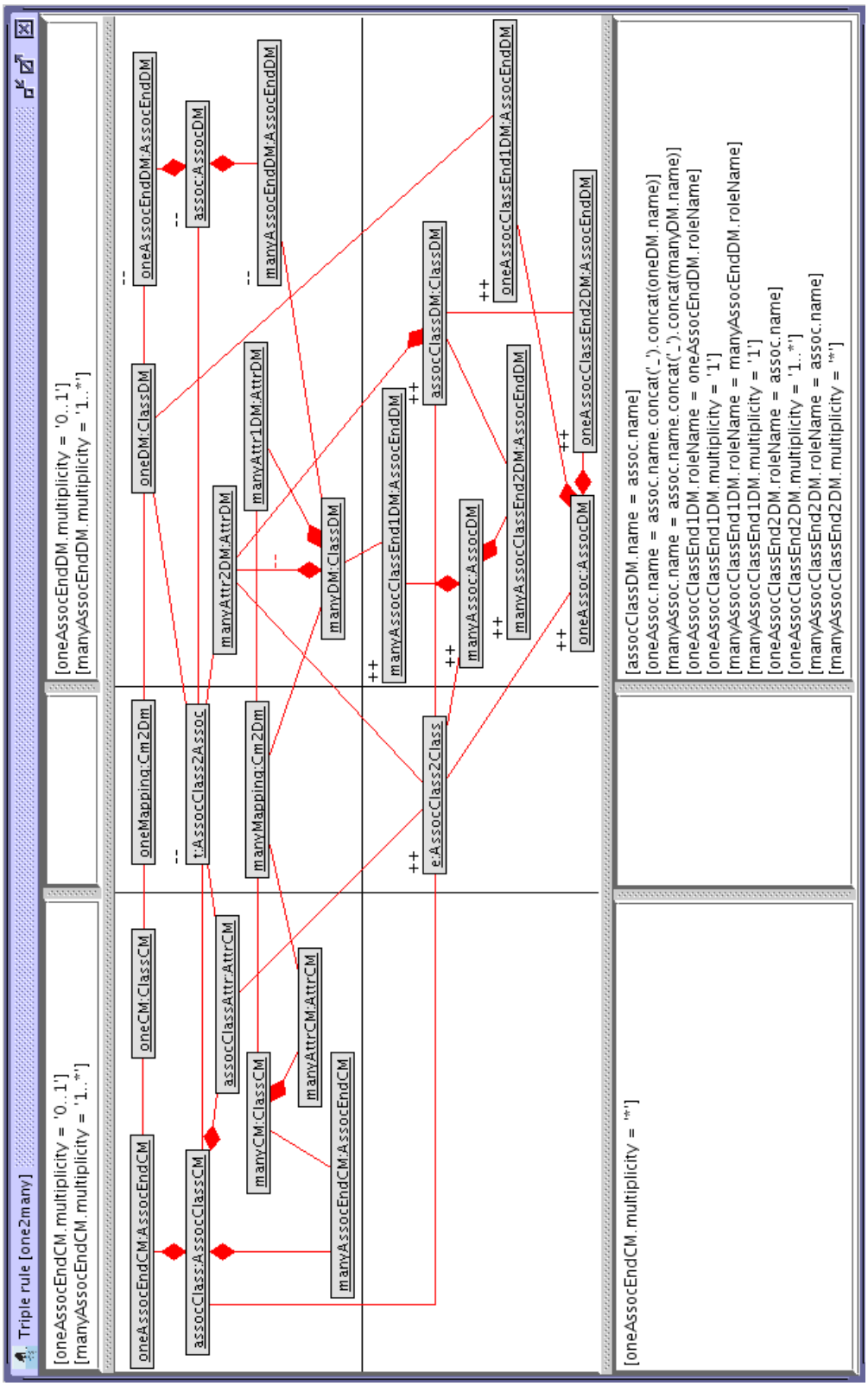


Figure 5.10: Triple rule `transAssocclass_one2many` including OCL conditions

tween the source and target metamodel by adding 'CM' or 'DM' into their UML class names, respectively.

Figure 5.10 shows the triple rule `transAssocclass_one2many` for the co-evolution step from the version `A` to the version `B` as shown in Fig. 5.8. This rule is presented in a compact form comparing to the form depicted in Fig. 4.6. Deleting and new nodes (or edges) are marked '−−' and '++', respectively. This triple rule can be realized by an OCL operation as explained in the section 5.2. The input of the operation is depicted in Fig. 5.6 and Fig. 5.7. The example transformation step by this rule is depicted as in Fig. 5.11.

5.3.3 Forward and Backward Transformation

A forward (backward) transformation based on triple rules is a transformation from the source (target) model to the target (source) model by a sequence of transformations that are carried out by derived triple rules for forward (backward) transformation. The derived triple rules for forward (backward) transformation are explained in Definition 3.17 and Definition 4.4.

Figure 5.12 presents an example for forward transformation. The note (1) in this figure depicts an evolution from the version `A` to the version `B*` when the association multiplicity in the conceptual model in the version `A` is changed. The design model in `A` needs to be updated so that the relationship between the conceptual and design models can be formally traced. In other words, we need to transform the conceptual model in `B*` into the design model. This transformation gives us an integration of the conceptual and design models. This integrated model can also be obtained by a model co-evolution step as illustrated in Fig. 5.8. Therefore, this transformation can be carried out by the triple rule derived from the triple rule `transAssocclass_one2many` for forward transformation. The note (2) in Fig. 5.8 depicts the application of this derived triple rule for an integrated model of the conceptual and design models in `B`. In this rule application the conceptual and design models in `A` correspond to the parts `SL` and `TL` of the triple `transAssocclass_one2many`, respectively. The conceptual model in `B*` corresponds to the part `SR` of the triple rule.

We can translate the derived triple rule for forward (backward) transformation into an OCL operation. The derived triple rule and its application conditions are completely defined from the original triple rule, see Definition 3.17 and Definition 4.4. Then, we organize the operation for the derived

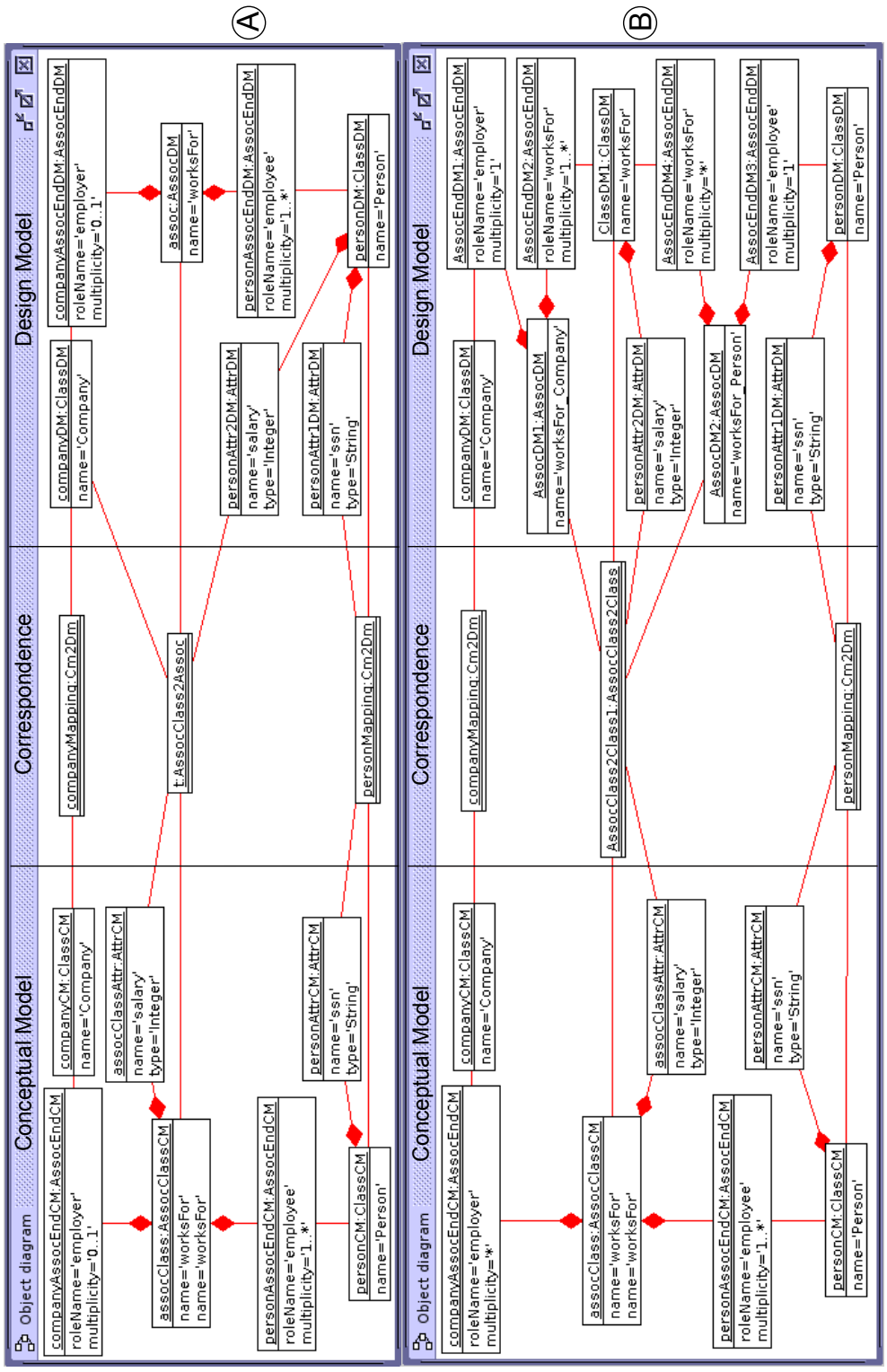


Figure 5.11: Co-evolution from the version `A` to the version `B` by the triple rule `transAssocclass_one2many`

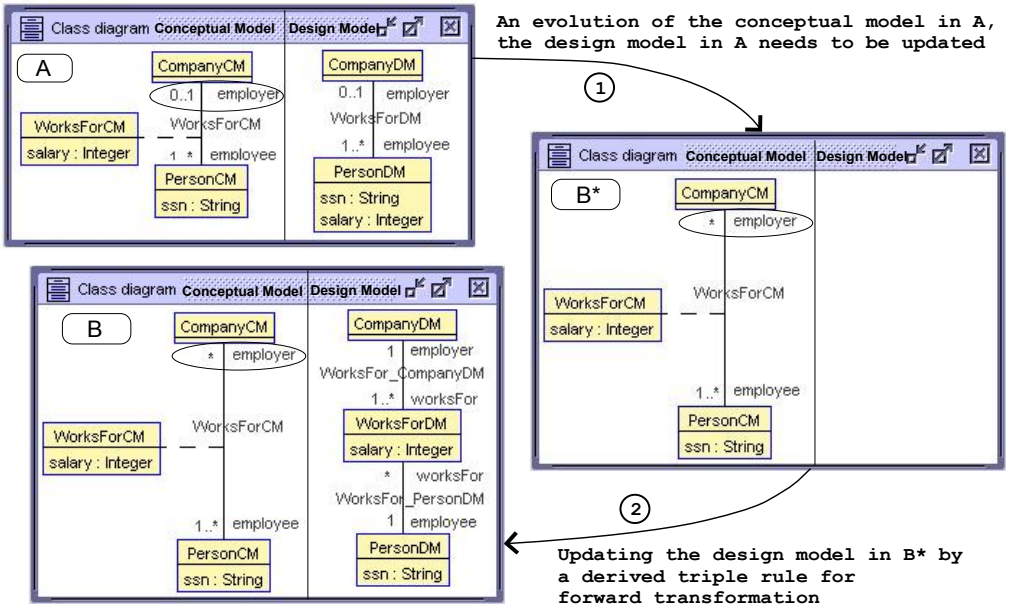


Figure 5.12: Forward transformation by derived triple rules

triple rule and its input as shown in Fig. 5.6. The input of the operation for forward (backward) transformation corresponds to cells denoted by 'I' in Fig. 5.6, i.e., it refers to the nodes of the patterns `SL`, `SR`, `CL`, and `TL` of the original triple rule. Since we have `SL`⊂`SR`, we can represent the input of this operation by parameters `matchSR`, `matchTL`, and `matchCL` of the Tuple type in OCL as shown in Fig. 5.7.

The operation for forward (backward) transformation can be realized similarly to the explanation in Sect. 5.2. For a declarative description of the operation, OCL pre- and postconditions are used as its operational contracts. For an operational description of the operation, USE command sequences are used to realize the operation. Note that the contract of the operation also includes OCL conditions which are defined by BAC^{trF}_{pre} and BAC^{trF}_{post} in Definition 4.4.

5.3.4 Model Integration

A model integration is a transformation to connect source and target models to each other by correspondence nodes, and this integrated model can also be generated by a triple graph grammar. For example, supposing that the trace between the conceptual and design models in B in Fig. 5.8 is damaged, we can employ a triple rule derived from the triple rule `transAssocclass_one2many`

for model integration to re-establish the trace.

The derived triple rule for model integration can be translated into an OCL operation. The derived triple rule and its application conditions are completely defined from the original triple rule, see Definition 3.17 and Definition 4.4. The operation for the derived triple rule can be designed as shown in Fig. 5.6. The input of this operation corresponds to cells denoted by 'I' in Fig. 5.6. The input of this operation refers nodes in the patterns `SL`, `SR`, `CL`, `TL`, and `TR` of the original triple rule. Since we have `SL`$\subset$`SR` and `TL`$\subset$`TR`, the input of this operation can be formed by parameters `matchSR`, `matchTR`, and `matchCL` of the Tuple type in OCL as shown in Fig. 5.7.

We can realize the operation for model integration in a similar way to the explanation in Sect. 5.2. For a declarative description of the operation, OCL pre- and postconditions are used as its operational contracts. The contract of this operation also includes OCL conditions which are defined by BAC^{trI}_{pre} and BAC^{trI}_{post} in Definition 4.4. For an operational description of the operation, USE command sequences are used to realize the operation.

5.3.5 Model Synchronization

The model transformation approach based on triple rules not only supports forward and backward transformations but also establish a correspondence between the source and target models. However, the correspondence may be damaged when the source and target models are changed. We propose new operations derived from triple rules in order to keep the source and target models consistent. The operations are referred to as the synchronization operations.

We focus on a specific situation of the inconsistency between source and target models. It is a change which can be located in the context of a triple rule application (together with a match). Moreover, only the part corresponding to the image of the RHS in the match is changed, the part corresponding to the LHS is unchanged.

For example, the note (1) in Fig. 5.13 shows a co-evolution transformation by the triple rule `transAssocclass_one2many` for an integration of the conceptual and design models in `B`. The note (2) shows a modification in the version `B`, resulting in the inconsistency in the version `C`. The first inconsistency is indicated by the dashed line in `C`. We recognize that there are two corresponding Attribute objects whose `name` attributes do not coincide because the value 'income' differs from 'salary'. This makes the invariant of the

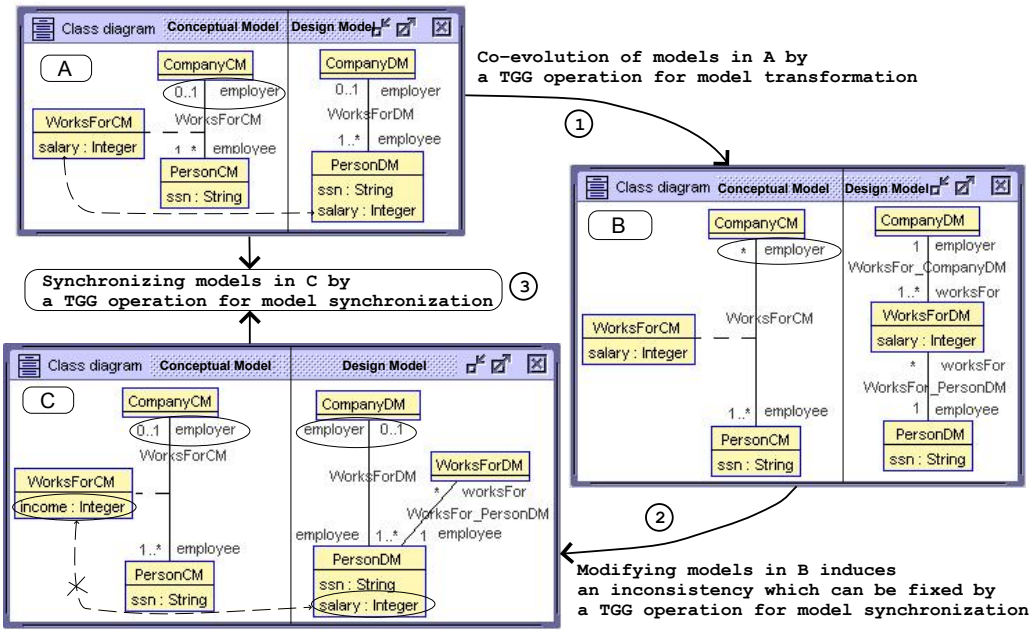


Figure 5.13: Detecting and fixing model inconsistency with TGG rules incorporating OCL

class `Cm2Dm` `self.attrCM.name=self.attrDM.name` not fulfilled. The other inconsistency can be recognized in `C` since object links are changed, resulting in the refinement of the association class not fulfilled.

In order to fix such an inconsistency, we can apply the synchronization operation of the corresponding triple rule. The note (3) in Fig. 5.13 depicts the synchronization in the version `C` by the synchronization operation of the triple rule `transAssocclass_one2many`. In the application of this synchronization operation, the versions `A` and `C` correspond to the LHS and RHS of this triple rule, respectively.

The synchronization operation is similar to the operation for model co-evolution. Unlike the operation for model co-evolution, this operation does not re-create objects that exist in the current model. Instead the objects are updated. The input of the synchronization operation is illustrated in Fig. 5.6 and Fig. 5.7. It includes two parts: The first part corresponds to the LHS of the triple rule, and the second part corresponds to the gluing part of the triple rule (including nodes which are in the RHS, but not in the LHS). The second part corresponds to objects that need to be updated when the operation is executed. The first part of the input is represented by parameters `matchSL`, `matchTL`, and `matchCL`, and the second part is represented by parameters `maskS`, `maskT`, and `maskC` as illustrated in Fig. 5.7.

The synchronization operation can be realized similarly to the explanation in Sect. 5.2. The pre- and postcondition of this operation are defined similarly to the one of the operation for model co-evolution. For an operational description of the operation, we can use USE command sequences to realize this operation.

5.4 Transformation Quality Assurance

For quality assurance of model transformations, we focus on the following questions: (1) Do models produced by transformations conform to metamodels?; (2) Are properties of the models fulfilled?; (3) Are transformations applicable, and are these applications correct?; and (4) Is the correspondence between source and target models in triple transformations kept?

This section shows that the frame based on TGG and OCL for model transformation, which can be implemented in OCL tools like USE, can support means for quality assurance of model transformation.

5.4.1 USE Support

With the tool USE [GBR07], UML class diagrams together with additional OCL constraints can be validated. Specifically, USE allow us to check class invariants, operation pre- and postconditions, and properties of models, which are expressed in OCL. In USE system states are represented as object diagrams. System evolution can be carried out using operations based on basic state manipulations, such as (1) creating and destroying objects or links and (2) modifying attributes. In this way the integration of TGGs and OCL are completely covered by USE.

Presenting models. Host models are represented as object diagrams in USE. The models are restricted by the metamodel invariants.

Matching TGG rules. Matching a rule is carried out by evaluating OCL queries on the source object diagram (working graph). These queries are captured by the precondition of the operation corresponding to the rule.

Applying TGG rules. A rule application is realized by an operation call. Applying the rule by USE commands realizing the rule, we create objects and links for the right-hand side. The sequence of rule applications can be presented by a sequence diagram.

5.4.2 Well-Formed Models

Models within the model-driven approach based on TGGs and OCL are defined by metamodels. OCL conditions are used for restrictions on metamodels. A model conforms to the metamodel only if these conditions are fulfilled. They are then referred to as well-formed models. Models produced by transformations may not be well-formed. For example, in [DGB07, VAB+08], we show that the resulting model of the transformation from UML activity diagrams to Communicating Sequential Processes (CSP) does not conform to its metamodel. Our OCL-based framework for model transformation allows us to check the well-formedness of models during a model transformation.

5.4.3 Checking Properties of Models

Models in transformations based on the integration of TGGs and OCL can be seen as object diagrams. Therefore, we can describe model properties as OCL conditions, and check these properties by querying the model (an object diagram) with OCL conditions.

5.4.4 Verification of Transformation

As pointed out in the section 5.3, transformation operations derived from a triple rule can be characterized by pre- and postconditions. Operation preconditions allow us to check if a rule is applicable. After each rule application, one may check the postconditions of the rule for an on-the-fly verification of the transformation.

5.4.5 Detecting and Fixing Model Inconsistency

Within the approach employing the integration of TGGs and OCL for model transformation, triple rules not only define a translation from one model to another, but also capture the correspondence between the source and target models for an integrated model. The correspondence is represented by links and OCL conditions so that it allows us to check and maintain the consistency between two models.

Detecting inconsistency. We consider an inconsistency in the integrated model when (`Class`, `Attribute` etc.) object attributes are changed. We

can detect the inconsistency by checking OCL constraints in the correspondence part of triple rules. For example, Fig. 5.13 shows such an inconsistency (indicated by the dashed line) in `C`: The `name` attributes of two corresponding `Attribute` objects do not coincide (the value 'income' differs from 'salary'), i.e., the invariant of the class `Cm2Dm` is not fulfilled: `self.attrCM.name=self.attrDM.name`. We consider another inconsistency when links in the integrated model are changed. The inconsistency can be detected in a semi-automatic way by querying models as the image of the triple rule from the host models using OCL conditions built by links and OCL constraints in the rule. In Fig. 5.13 models in the version `C` can be queried using OCL conditions derived from the TGG rule `transAssocclass_one2many`. In the version `C` we can recognize an inconsistency since object links are changed inducing that the refinement of the association class is not fulfilled.

Fixing inconsistency. Detecting inconsistency in the way pointed out above allows us to fix the inconsistency manually. We can employ the operation for mode synchronization in order to fix the inconsistency. The operation will update objects and links in the target version. In Fig. 5.13, see the note (3), the versions `A` and `C` correspond to the LHS and RHS of the triple rule. The synchronization of the version `C` allows us to fix the inconsistency.

5.5 Related Work

Beside approaches based on QVT as a standard offered by OMG for model transformation, there are many different approaches for model transformation [CH03] such as approaches based on XSLT [1] or graph transformation. Our approach employing the integration of TGGs and OCL for model transformation is not only based on the theoretical work on graph transformation, but can also be compared to approaches realizing QVT such as with the tool VMTS [LLVC06]. Our approach and QVT realizations are also toward an OCL-based framework for model transformation [GKB08].

Model transformation based on TGGs has been implemented in Fujaba [KW07] by two different approaches. With the first approach for a TGG-Compiler, triple rules are translated into Java in order that they can be executed by Java programs. With the second approach for a TGG-Interpreter [KRW04], the execution of triple rules is implemented on the Eclipse Modeling Framework[2]. In contrast to these approaches, we real-

[1] see www.w3.org/TR/xslt

[2] see www.eclipse.org

ize triple rules including OCL conditions in an OCL-based framework for model transformation with the support of USE [GBR07]. Our OCL-based framework supports a quality assurance for model transformation. Moreover, within our approach a new operational scenario derived from triple rules for model synchronization is proposed and realized.

In [GBD08, BG06] an OCL-based realization in USE for graph transformation is proposed. That is an alternative approach for graph transformation engines, which can be compared to the current approaches such as AGG [dLBE+07], Fujaba [Wag06], GROOVE [KR06], and Progres [SWZ96]. While that work can be seen as the modeling based on UML and OCL for graph transformation systems, our current work focuses on modeling triple graph transformation systems, which correspond to transformation scenarios including forward and backward transformation, model integration, and model synchronization.

Regarding to the translation of graph rules into other domains, the work in [VFV06] proposes employing a relational database in order to implement graph transformations. In [CCGdL08], graph rules are also translated into OCL-based representation in order to analyze properties of transformations. The basic idea for the analysis is to establish a transformation model for transformations [BBG+06]. In [BS06] graph rules are translated into Alloy for the same goal.

In [GW06] an algorithm to locate model inconsistency within integrated models (by triple transformations) is proposed. The core of the algorithm is backtracking the triple derivation for the integrated model, and then reapplying the forward or backward transformation operation in order to fix the inconsistency. Within our approach, the inconsistency is located by checking OCL conditions within the correspondence part. Then, we employ the operation for model synchronization in order to fix the inconsistency.

Our work can be seen as an approach to model traceability [ARNRSG06]. The TGG approach supports an interesting feature for the motivation: Trace links can be automatically generated. In [FV07], the approach to tracing two models by a middle model, a so-called weaving model, is considered. We can see TGGs as the very transformations on weaving models. OCL constraints within our integration of TGGs and OCL are the very constraints for correspondences within weaving models.

Summary

In this chapter, we have proposed an approach for an operational description of model transformation based on triple rules incorporating OCL. Operational scenarios of triple rules incorporating OCL are realized towards an OCL-based framework for model transformation. Triple rules incorporating OCL can be translated into operations for realizing transformation scenarios: The OCL operations allow us to execute forward- and backward transformations, model co-evolution, model integration, and model synchronization.

The language USE4TGG allows us to present triple rules incorporating OCL. The USE4TGG description of triple rules is also used to realize transformation operations. The OCL operations are realized by taking two views on them: Declarative OCL pre- and postconditions are employed as operation contracts, and command sequences are taken as an operational realization.

Our approach has been realized based on the tool UML-based Specification Environment (USE). USE offers full OCL support for restricting models and metamodels with invariants, for checking pre- and postconditions of operations as well as for validating and animating transformation operations. Our OCL-based framework for model transformation together with the USE support offers means for quality assurance of transformations such as to check well-formedness constraints and properties of models, to verify transformations, and to detect and fix model inconsistency.

Chapter 6

Towards Precise Operational Semantics of Use Cases

Use cases have achieved wide acknowledgement for capturing and structuring software requirements. The informality of use cases is a barrier from integrating them into model-driven engineering. This chapter proposes using TGGs and OCL in order to describe operational semantics of use cases in a precise way. The core of this approach is to relate scenarios at different levels of abstraction by triple rules incorporating OCL.

6.1 Introduction

Use cases have achieved wide acknowledgement for capturing and structuring software requirements. The UML2 specification [OMG07c] refers to them as the central point of modeling the behavior of a system, a subsystem, or a class. Use cases are typically represented as a combination between an informal UML use case diagram and loosely structured textual descriptions. The informality of use cases is a barrier from integrating them into model-driven engineering. This chapter proposes an approach based on TGGs and OCL in order to precisely describe operational semantics of use cases.

There are several ways to define use case [Jac92, RJB04, Coc00]. The work in [RJB04] defines use case as follows: "A use case is the specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside objects to provide a service of value". Many researches as surveyed in [Hur97] have worked in order to introduce rigor into use case descriptions, ranging from the syntax aspect to the semantics aspect. The work in [Sen02, Coc00] proposes a precise structured text for use case descriptions. The work in [SBN+07, DLMP04, RB03] proposes metamodels in order to form a conceptual frame for use cases. Many papers propose to use activity diagrams [AJI04, LS06] or statecharts [SPW07, NFTJ06, GLST01] in order to specify them.

In this chapter we propose another approach for describing the semantics of use cases in a precise way. The core of our approach is to describe an integrated view of use case models and design models in order to achieve the operational semantics of use cases. Specifically, our goals are as follows:

- Establish views capturing the system execution by snapshots at the use case and design levels.
- Develop an operational mechanism to describe the semantics of the integration by tracing the execution of the system at the two levels.

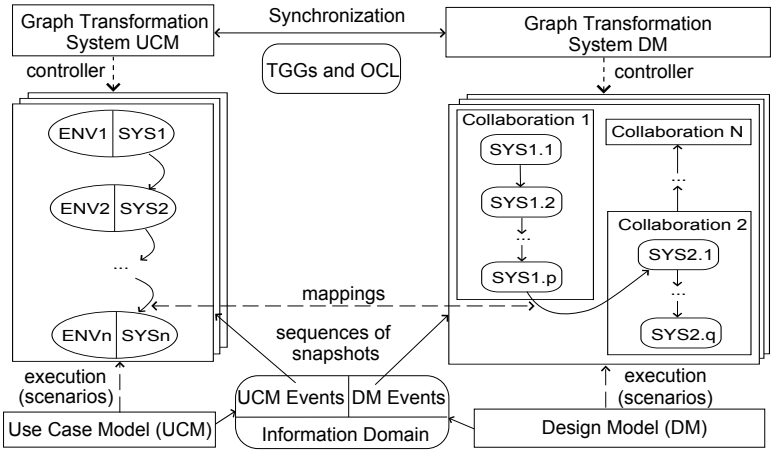


Figure 6.1: Overview of the approach

Our approach for describing the operational semantics of use cases based on the integrated view of use case models and design models can be illustrated as in Fig. 6.1. On the left side, the use case model allows us to capture system executions as sequences of use case snapshots. Use case snapshots are denoted by ovals. ENV and SYS denote the state of the input (environment) events and the system state, respectively. On the right side, snapshot sequences reflecting a corresponding system execution at the design level are presented. We develop metamodels for use case models and design models based on the UML metamodel. The metamodels are extended by graph transformation rules, which allow us to define use case scenarios. In this way, snapshot sequences can be controlled by a graph transformation system at each level. We define triple rules incorporating OCL in order to relate the graph transformation systems and to synchronize scenarios at these levels. By executing the triple graph transformation system, an integrated semantics of use case models and design models can be obtained, resulting in an operational semantics of use cases.

Our mechanism for the operational semantics of use cases offers many benefits. On the one hand, this mechanism allows us to generate scenarios as test cases, to validate system behavior, and to check the conformance between use case models and design models. This supports basic constructions of an automatic and semi-automatic design. On the other hand, this mechanism opens the possibility to simulate the automatic evolution of the system so that we can check properties as well as non-functional requirements, e.g., performance or security features of the designed system.

The rest of this chapter is organized as follows. Sections 6.2 and 6.3 explain metamodels for use case and design models. Section 6.4 defines triple rules incorporating OCL for the operational semantics of use cases. Section 6.5 explains how scenarios are synchronized for a precise operation semantics. Section 6.6 states the USE-based implementation of our approach. Section 6.7 discusses on related work. This chapter is closed with a summary.

6.2 Metamodel for Use Cases

This section defines a metamodel for use case models. This metamodel is an extension of the UML metamodel. For this metamodel, we reuse concepts in the UML metamodel of use case and activity diagrams. The other concepts are defined based on our approach to use cases.

6.2.1 Example Use Case

Figure 6.2 presents an example use case model. This use case model can also be presented in a UML use case diagram as shown in Fig. 6.3. This use case model describes (fragments of) the service of a car rental system in a textual format. Let us start with the use case `Return Car`. The textual description of this use case states the general information including the actor, goal, trigger, and pre- and postconditions. The basic and alternate flows and extensions of this use case show scenarios of using this service.

In the example use case model the `include`, `extend`, and `generalization` relationships between use cases are illustrated. First, the `Return Car` use case includes the `Process Payment` use case since `Return Car` refers to that use case at the *inclusion point*, step (7) of the basic flow. This flow rejoins at step (8) as soon as the corresponding scenario in that new use case is finished. Second, the `Return Car` use case can be extended by the `Handle Late Return` use case. Once the *extension point*, i.e., step (6) of the basic

flow of the `Return Car` use case, is defined, this flow transfers to the scenario of the new use case and rejoins at the next step of the extension point. When the flow of a use case reaches a step referenced by the extension point and the condition of the extension is satisfied, the flow will transfer to the behavior sequence of the extension use case. When the execution at the extension use case is complete, the flow rejoins the original use case at the referenced point. Finally, we have a generalization relationship between the `Process Payment` use case and the `Process Credit Payment` use case. The former inherits from the previous one since the actions (1) and (2) in the basic flow of the `Process Credit Payment` are a refinement of the action (1) in the `Process Payment` use case.

Use Case: Return Car
Actor: Clerk
Goal: To process the case when a car is returned.
Trigger: Customer wants to return a car.
Precondition: the rental exists and the car was delivered.
Postcondition: the rental is closed and the car is available.
Basic Flow:
1. Clerk requires to process a rental.
2. System asks the customer id.
3. Clerk enters the id.
4. System displays the rental.
5. Clerk enters the mileage.
6. System updates the fee.
7. ***Include Process Payment***.
8. System closes the rental.

Alternate Flows:
4.a. The rental is not found.
 4a1. System informs that the rental does not exist.
 4a2. Return to the step 3 of the Basic Flow.

Extensions:
E1. *Late Return*:
The extension point occurs at the step 6 of the Basic Flow.

Use Case: Process Payment
Actor: Clerk
Basic Flow:
1. System handles the payment.
2. System prints the invoice.

Use Case: Process Credit Payment
Actor: Clerk, Card Reader
Basic Flow:
1. Card Reader reads the credit card information.
2. System updates the rental for the payment. (*steps 1-2 refine step 1 of Process Payment*)
3. System prints the invoice.

Use Case Handle Late Return
Actor: Clerk
Basic Flow: (none)
Alternate Flows: (none)
Extension Flows:
EF1. *Process Later Return*:
This extension flow occurs at the extension point *Late Return* in the Return Car use case when the customer returns a car late.
1. System updates the rental for the return late case.
2. System rejoins at the extension location.

Figure 6.2: Use case description in a textual template format

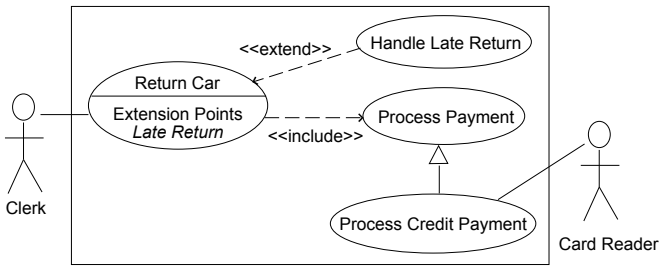


Figure 6.3: UML use case diagram for the example use case model

6.2.2 Concepts for the Use Case Metamodel

In order to obtain a precise description of use cases, we re-express the textual description of use cases, e.g., the description shown in Fig. 6.2, in terms of design models, e.g., the conceptual model presented in Fig. 6.4. This allows us to conceptualize a new view of use cases as follows.

Use case scenarios. A use case scenario is a state sequence of the interaction between the actor and the system, starting when the use case is invoked, ending when it finishes. It is also referred to as an instance of the use case.

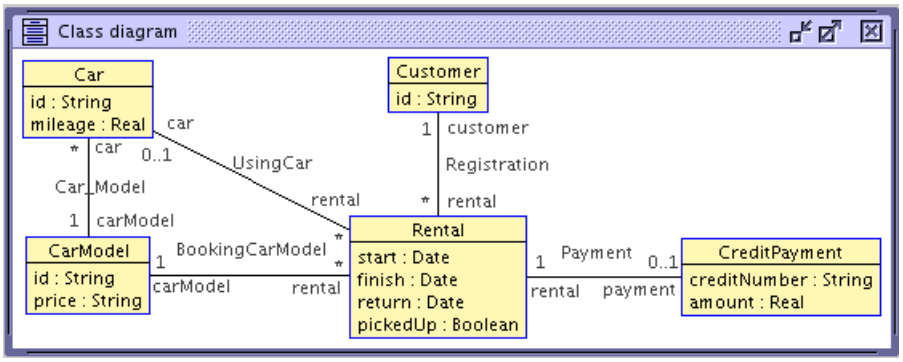


Figure 6.4: Conceptual model in the case study

Use case snapshots. We use concepts of the conceptual domain in order to express the interaction state of use case scenarios. The conceptual model within the case study is shown in Fig. 6.4. Each interaction state is referred to as a *use case snapshot*. A use case snapshot reflects both the system state and the state of the input events. They can be represented in an object-diagram-like form including objects, links, and OCL conditions.

Actions. Use cases can be presented by activity diagrams. Each step in the use case flow corresponds to an action in the activity diagram. Each action can be carried out by the actor, the system, or the scenario of the including or extending use cases. This corresponds to three kinds of actions: *system action*, *actor action*, and *use case action*. Actions at the use case level may play the role as decision nodes in activity diagrams, and they will be referred to as conditional actions.

Action contracts. We employ contracts which are pairs of pre- and post-conditions in order to express the effect of actions. The pre- and postconditions for a contract are represented by use case snapshots.

Flow conditions. Flow conditions are used in the following situations: (1) When an exception arises from a system action, (2) when the flow trans-

fers to another use case at the extension point, or (3) when the actor selects an option. Flow conditions are also represented by use case snapshots.

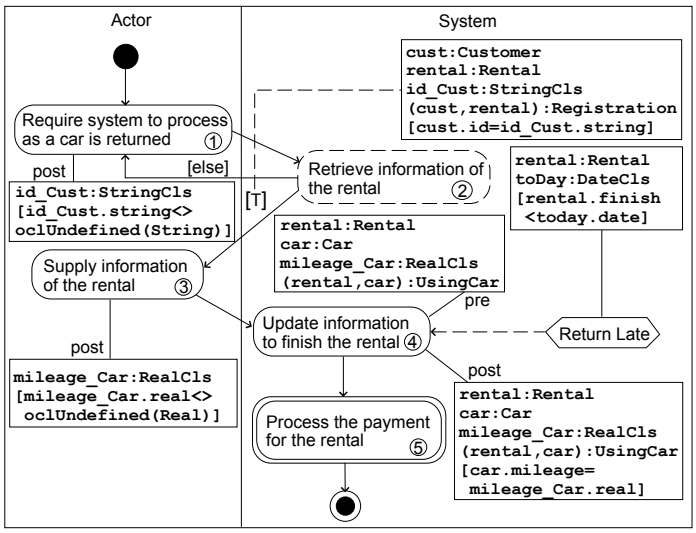


Figure 6.5: Represent the `Return Car` use case in terms of design models

Figure 6.5 shows an extended activity diagram for presenting the `Return Car` use case in a precise way. In this diagram, use case snapshots are denoted by rectangles. System and actor actions such as the actions (1) and (4) are denoted by rounded rectangles. Use case actions such as the action (5) are denoted by the rounded double-line rectangles. A conditional action such as the action (2) is denoted by a rounded dashed-line rectangles. The flow condition `[T]` is attached with a use case snapshot. Extension points such as the `Return Late` extension point of the action (4) are denoted by six-sided polygons. The classes including `StringCls`, `RealCls`, and `DateCls` are used as wrappers for primitive data types, respectively. The aim is that snapshots as pre- and postconditions of actions can be represented as object diagrams.

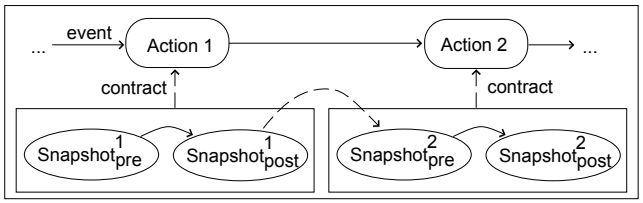


Figure 6.6: Representing snapshot sequences by actions and contracts

Figure 6.6 illustrates how we can use the extended activity diagram to represent a use case as a set of scenarios, while each scenario is a sequence of use case snapshots.

6.2.3 Presentation of the Use Case Metamodel

Figure 6.7 pictures the use case metamodel. Concepts and relationships highlighted by the bold lines do not come from the UML metamodel but are newly proposed in our approach. The other concepts belong to the UML metamodel [OMG07b, OMG07c].

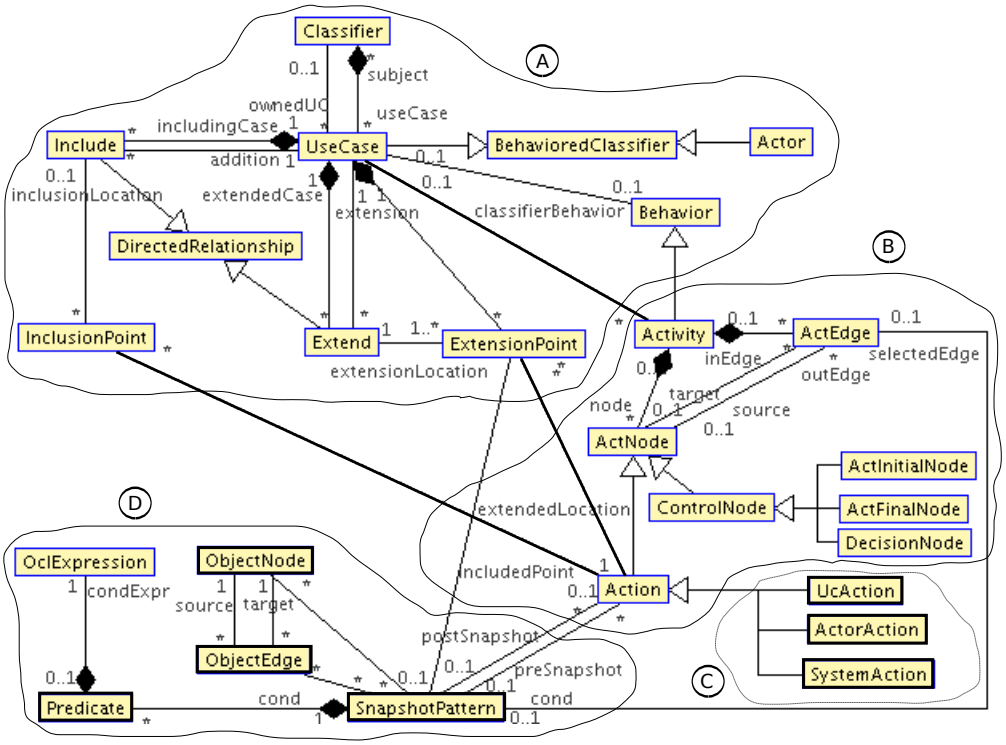


Figure 6.7: Metamodel for use cases

This metamodel can be divided into four groups. The group `A` includes concepts for use case diagrams. This group belongs to the `UseCase` package of the UML metamodel. The main concepts in this group include `UseCase`, `Include`, `Extend`, `InclusionPoint`, and `ExtensionPoint`. The concepts have been explained in Subsect. 6.2.1. Let's consider the other concepts in this group. Since the inclusion and extension relationships are the directed relationships, we have a generalization from `Include` and `Extend` to `DirectedRelationship`. A `Classifier` represents the subject to which the owned use cases apply. The same use case can be applied to multiple subjects, as identified by the `subject` association role of a `UseCase`. A `UseCase` represents a declaration of an offered behavior that the subject can perform in collaboration with one or more actors. Therefore, a `UseCase` is a kind of behaviored classifier which is represented by the concept `BehavioredClassifier`,

and a `Use Case` is associated to a `Behavior`.

The group `B` includes basic concepts for UML activity diagrams. This group belongs to the `Activity` package of the UML metamodel. The group `C` consists of concepts for specifying actions in use case descriptions. Finally, the group `D` includes concepts for specifying use case snapshots. Basically the concepts in group `C` and `D` are new (except the `OclExpression` concept).

The association between the `UseCase` and `Activity` metaclasses means that use cases can be represented by activity diagrams. The association between the `Action` metaclass and the `InclusionPoint` (`ExtensionPoint`) metaclass represents the `Include` (`Extend`) relationship between use cases.

The `SnapshotPattern` metaclass represents use case snapshots. A `SnapshotPattern` object consists of the `ObjectNode` and `ObjectEdge` objects. An `ObjectNode` object corresponds to a domain class in the conceptual class diagram. An `ObjectEdge` object corresponds to a link between two `ObjectNode` objects.

A `SnapshotPattern` object may express pre- and postconditions of actions when it is linked to the corresponding `Action` objects. A `SnapshotPattern` object can also be the condition of extension points when it is linked to `ExtensionPoint` objects. A `SnapshotPattern` object will be the branch condition when it is connected to the `ActEdge` objects.

6.3 Metamodel for Design Model

We define a metamodel to present the execution scenario of the system at the design level. We aim to relate system snapshots at the design level to snapshots at the use case level in order to illustrate the semantics of use cases. This metamodel allows us to completely specify the system behavior, i.e., snapshots at the use case and design levels are completely defined when these two views of the same system execution are combined.

6.3.1 Concepts for the Metamodel

We extend activity diagrams in order to represent scenarios as sequences of system snapshots. In this way the metamodel for scenarios at the design level is an extension of the UML metamodel for activity diagrams. Figure 6.8 presents scenarios at the design level for the `Return Car` use case in an extended activity diagram.

Actions in scenarios at the design level are organized in a hierarchy using action groups. This hierarchy originates from a mapping between a sequence diagram and a corresponding extended activity diagram: The interaction sequence between objects (by messages) is represented by action sequences. Each message sent to a lifetime line in the sequence diagram corresponds to an action or an action group which realizes the object operation invoked by this message. The action group includes actions and maybe other action groups. An action group is always linked to an object node at the corresponding lifetime line. For example, Fig. 6.9 shows the sequence diagram that can be represented by the extended activity diagram as pictured in Fig. 6.8.

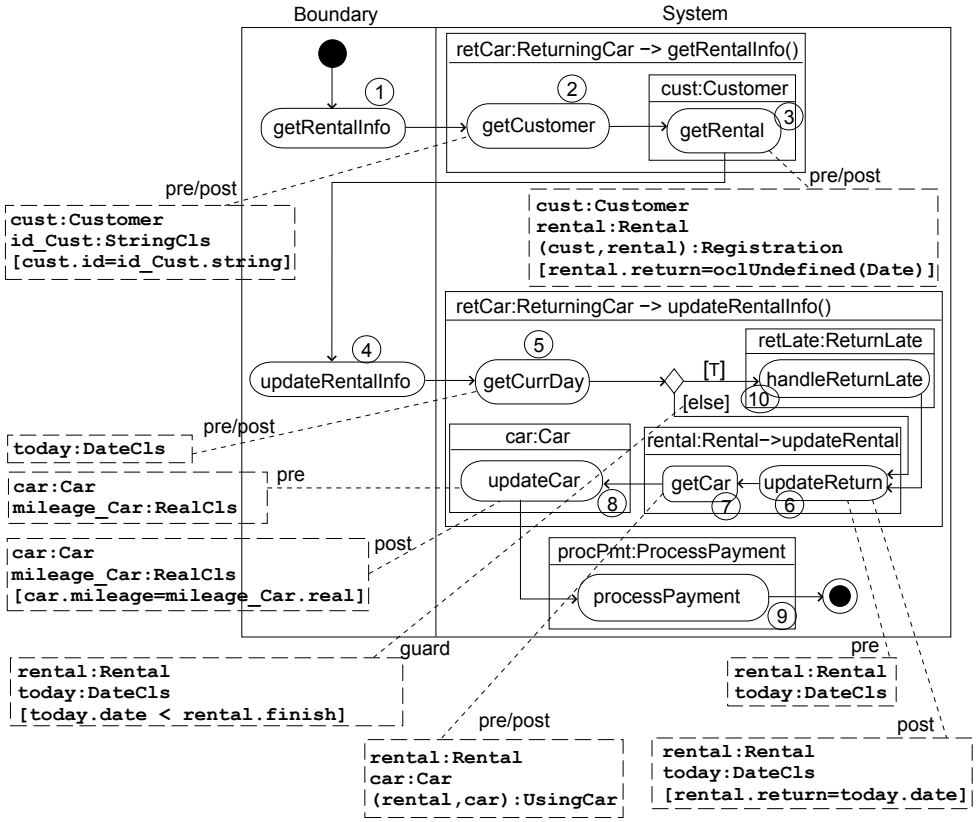


Figure 6.8: Extended activity diagram for presenting design model scenarios

6.3.2 Presentation of the Metamodel

Figure 6.10 presents the DML metamodel for scenarios at the design level. The concepts grouped in this metamodel belong to the `Activity` package of the UML metamodel. The remaining concepts, highlighted by bold lines, are used in order to represent snapshots at the design level.

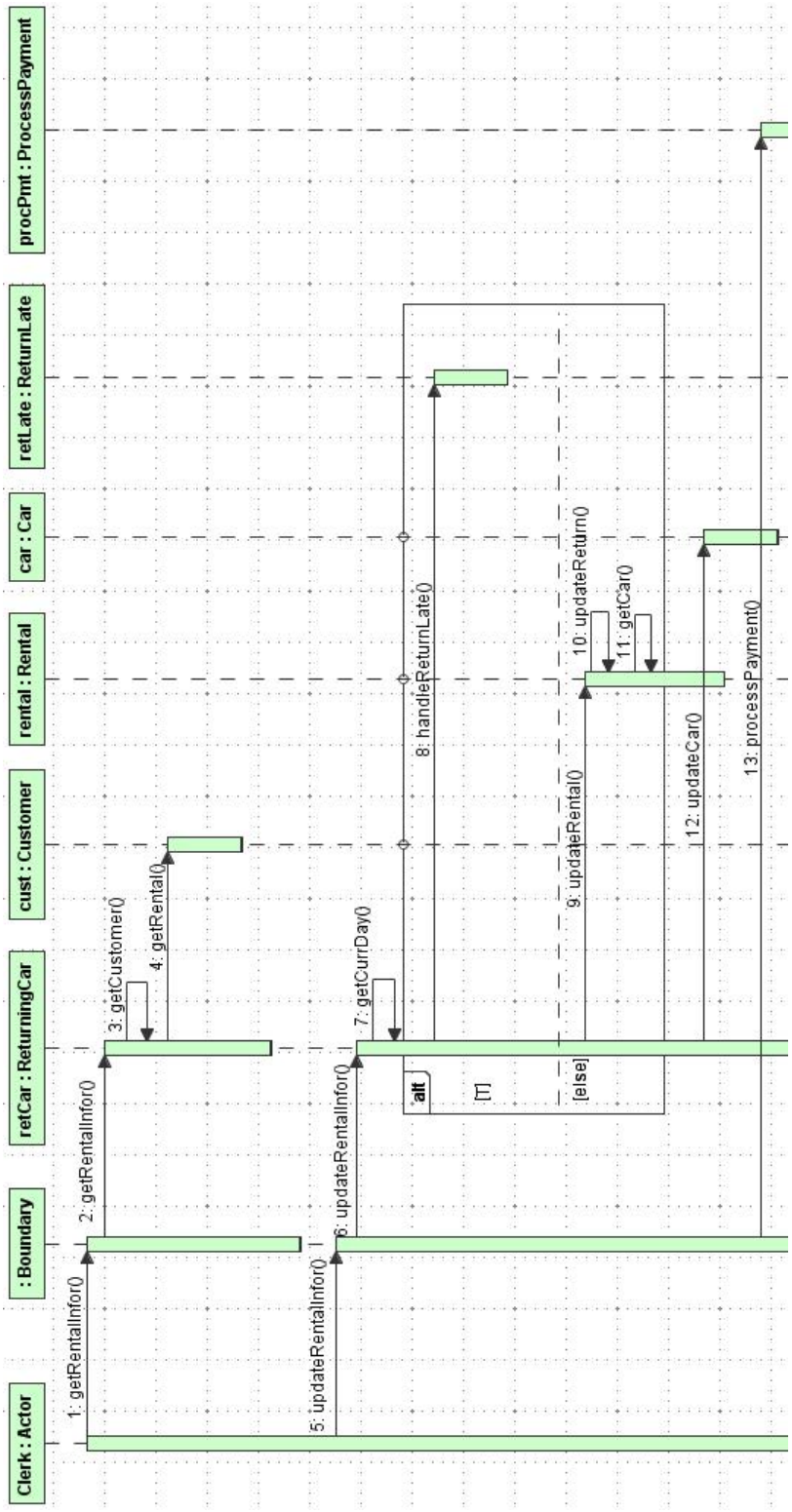


Figure 6.9: Sequence diagram realizing the example use case

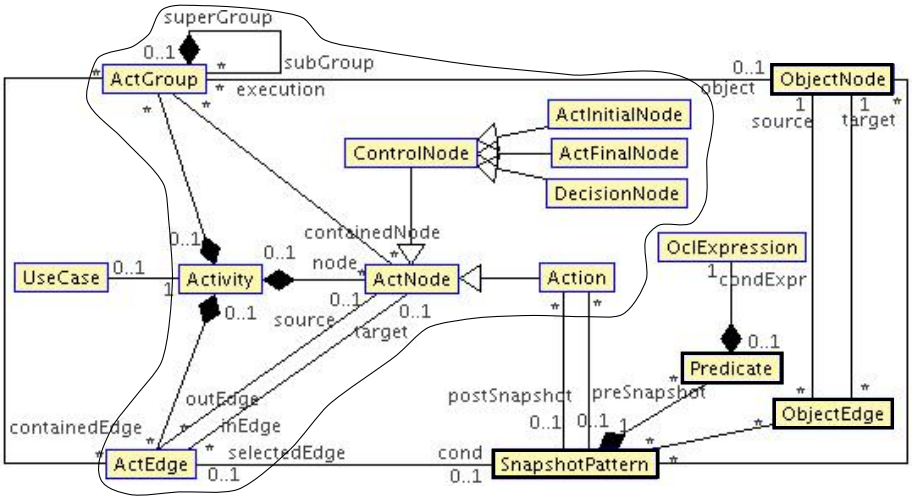


Figure 6.10: The metamodel for design-model scenarios

The `ActGroup` metaclass represents action groups. The `subGroup` association represents the hierarchy of actions. This metaclass is connected to the `ObjectNode` metaclass, so that action groups can be mapped to behaviors of objects. The remaining metaclasses represent snapshots. They are defined similarly to the ones for snapshots at the use case level.

6.4 TGGs and OCL for Use Case Semantics

In this section we define triple rules in order to integrate scenarios at the use case and design levels. A pair of corresponding scenarios supports two views on the same system execution. This results in a description of the operational semantics of use cases.

6.4.1 Co-evolution of Snapshots by Triple Rules

Figure 6.11 presents a co-evolution step of system snapshots at the use case and design levels for an execution scenario. The left side marked by (1) depicts the current actions at the use case and design levels. Snapshots as the postcondition of these actions are the current snapshots of a snapshot co-evolution for an execution scenario. A co-evolution step of snapshots is carried out when the next actions are 'added' and defined as shown on the right side (marked by (2)). Snapshots as the postcondition of the next actions will be the current snapshots for the execution scenario.

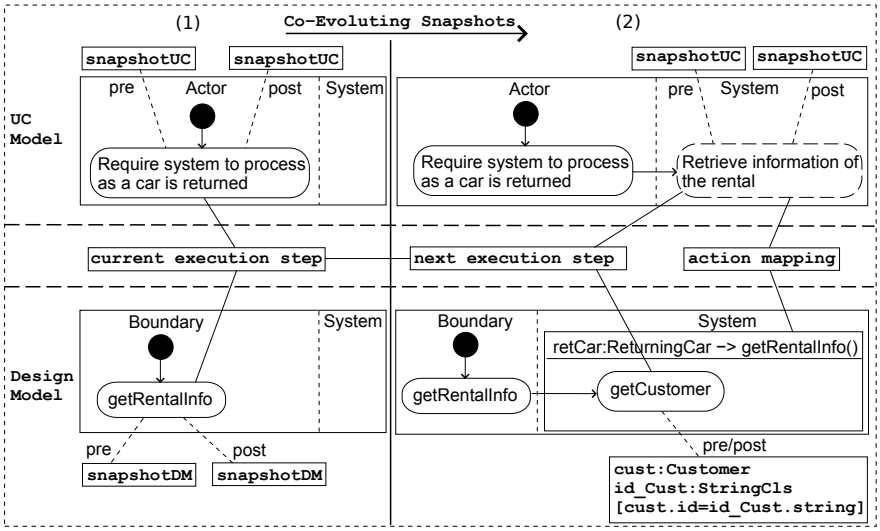


Figure 6.11: Co-evolution steps of snapshots for an execution scenario

Triple rules incorporating OCL allow us to express such co-evolution steps. The state of the system execution viewed from the use case and design levels can be represented by a triple graph. The source and target parts of this triple graph represent scenarios as action sequences together with snapshots (for pre- and postconditions and guard conditions). The correspondence part of this triple graph includes mappings between these scenarios. At the initial state we have the correspondence between the initial actions of the activity diagrams for the use case and design models. At each execution step we have the correspondence between current actions at these levels. An action at the use case level often corresponds to many actions at the design level within a refinement of use cases.

6.4.2 Defining Triple Rules Incorporating OCL

For each transfer step in the system execution at the use case and design levels, we need (1) to check the precondition of the next action at each level, (2) to perform the next action at the design level, or the actor action at the use case level, and then (3) to check the postcondition of the next action at each level. The precondition of actions at the use case level is checked only if object nodes in the corresponding snapshot are completely defined.

We define triple rules based on situations of action transfers in activity diagrams at the use case and design levels. At the use case level we have transfers for the next actor action, the next system action, the next use case

action, and the next action in the extending use case. These transfers can be with or without guard conditions. At the design level we have transfers for the next action in the same action group, and the next action in a new action group. These transfers can also be with or without guard conditions. Specifically, we define triple rules for this co-evolution as follows.

R1. Triple rule to start the scenario. The rule shown in Fig. 6.12 allows us to start the integration of scenarios at the use case and design levels. The `Act2Act` object in the LHS of this rule indicates the correspondence between the activity diagrams at these levels. Snapshots as the precondition of the use case scenario (on the left side) and the design scenario (on the right side) need to be fulfilled. This is expressed by the corresponding OCL conditions.

When this rule is applied the initial actions of these activity diagrams become the current actions in this execution. They are linked to the current `ExecControl` object.

Example. The integration of scenarios depicted in Fig. 6.5 and Fig. 6.8 can be started by applying this rule. The result is that two initial actions in these activity diagrams are marked as the current actions.

R2. Triple rule for the next actor action. The rule depicted in Fig. 6.13 is applied when the next step of the system execution is an actor action at the use case level. This application represents the situation when the system asks the actor to define the input value. This input value becomes the input for actions at the design level.

The OCL condition in the correspondence part of this rule ensures that the considered actions are the current actions. OCL conditions in the use case (source) and design (target) parts check whether the performance of the current actions at the use case and design levels is valid.

When this rule is applied the next actor action is assigned to the snapshots as pre- and postconditions. This action corresponds to the action of the boundary class at the design level. By performing these actions the input value from the actor can be defined. The postcondition of these action is checked in the next rule application.

Example. This rule is applied when the next step of the execution refers to the action (1) as shown in Fig. 6.5 and the action (1) as shown in Fig. 6.8. These actions correspond to the RHS of this rule. The initial actions correspond to the LHS of this rule.

R3. Triple rule for the next actor action with guard conditions. The rule pictured in Fig. 6.14 is applied when the next execution step is an

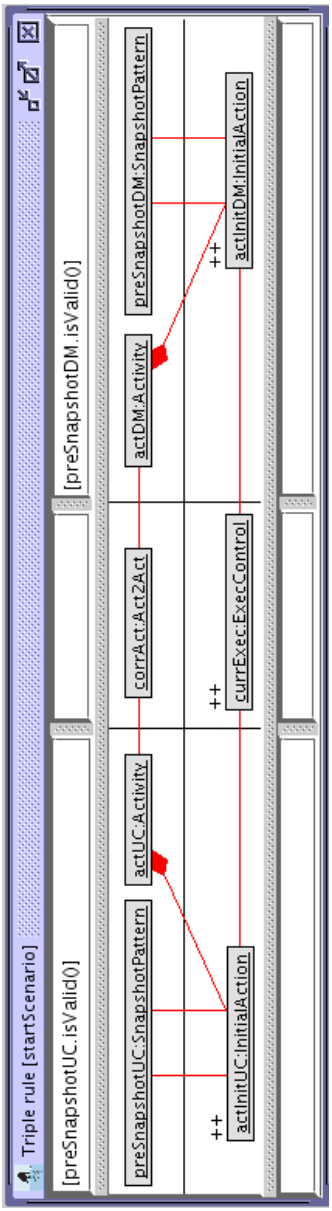


Figure 6.12: Triple rule to start the scenario

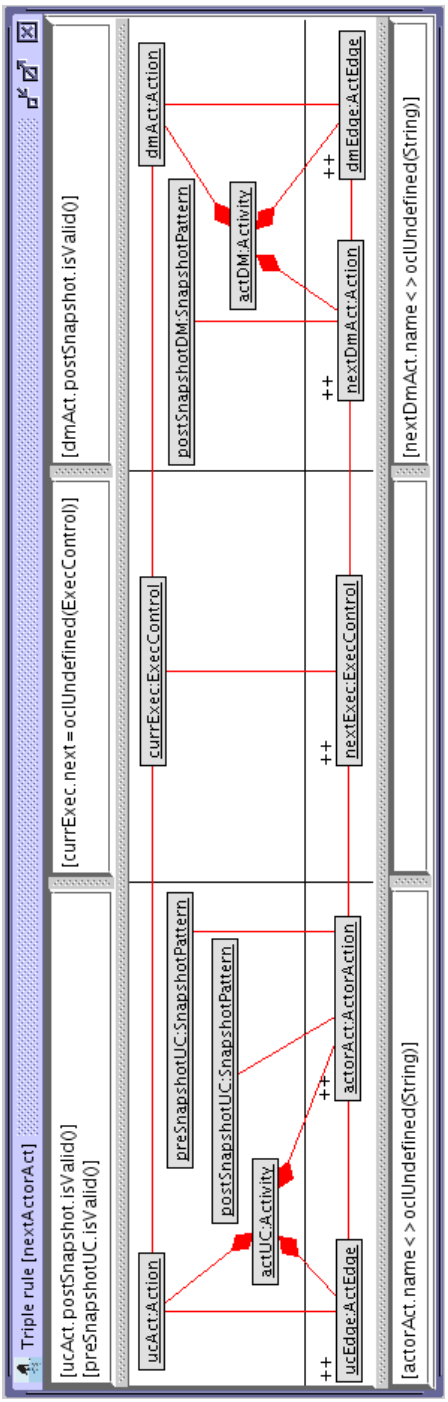


Figure 6.13: Triple rule for the next actor action

actor action. This rule is similar to the `R2` rule. The only difference between them is that before the system execution transfers to this actor action, a guard condition needs to be checked. This guard condition is represented by the `guardSnapshotUC` shown in Fig. 6.14.

Example. This rule is applied for the transfer from the action (2) to the actor action (3) as shown in Fig. 6.5 and the corresponding transfer from the action (3) to the action (4) as shown in Fig. 6.8.

R4. Triple rule for the next system action. The rule depicted in Fig. 6.15 is applied when the next execution step is a system action at the use case level. This transfer induces the invocation of a new action group at the design level.

OCL conditions in the LHS of this triple rule allow validating snapshots and ensuring the considered actions to be the current actions. OCL conditions in the RHS allow checking attribute values.

When this rule is applied, the system action `sysAct` in the use case part of this rule and the corresponding action group `actGrp` in the design part are created. The correspondence between them is represented by links that are connected to the `SysAct2ActGrp` object. The precondition of these actions is checked before this rule application, and the postcondition is checked in the next rule application.

The link between `actGrp` and `objNode` indicates that this action group is realized by the operation of this object.

Example. This rule is applied for the transfer from the action (3) to the system action (4) as presented in Fig. 6.5 and the corresponding transfer from the action (4) to the action (5) as presented in Fig. 6.8.

R5. Triple rule for the next action at the design level. The rule shown in Fig. 6.16 is used for the transfer from the current action at the design level to the next action that is in the same action group. There is no transfer at the use case level in this execution step since these actions are part of an action sequence which realizes the corresponding system action at the use case level.

Example. This rule is applied for the transfer from the action (6) to the action (7) as presented in Fig. 6.8. The current action at the use case level is the action (4) as shown in Fig. 6.5. This action is still the current action after this execution step.

R6. Triple rule for the next action with guard conditions at the design level. The rule pictured in Fig. 6.17 is used for the transfer from

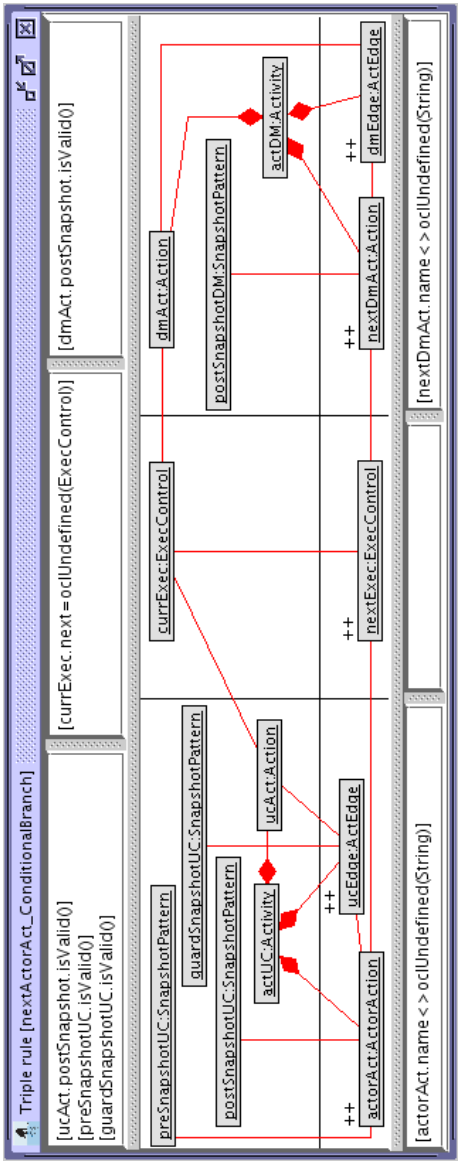


Figure 6.14: Triple rule for the next actor action with guard conditions

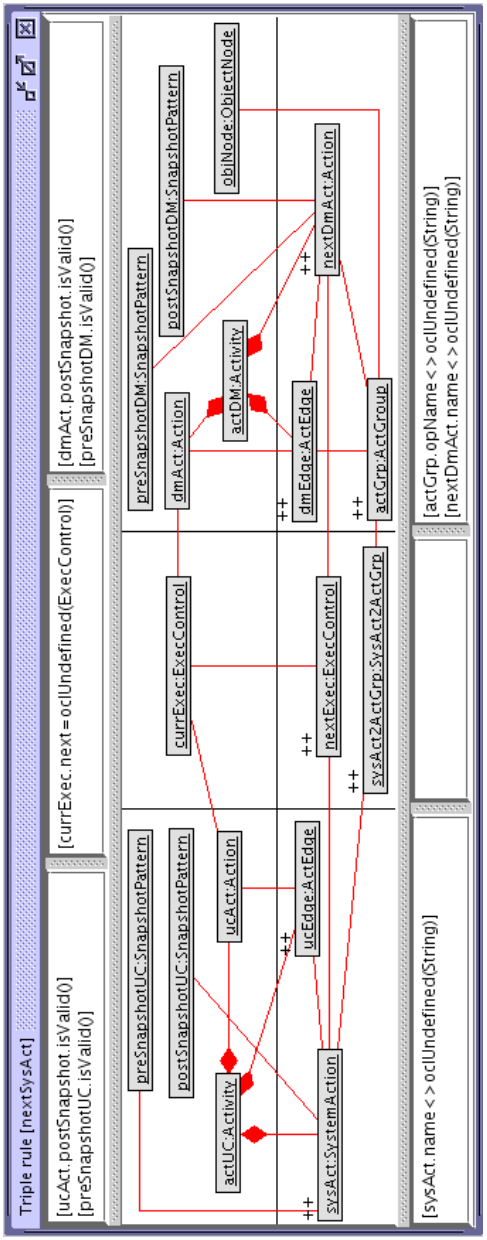


Figure 6.15: Triple rule for the next system action

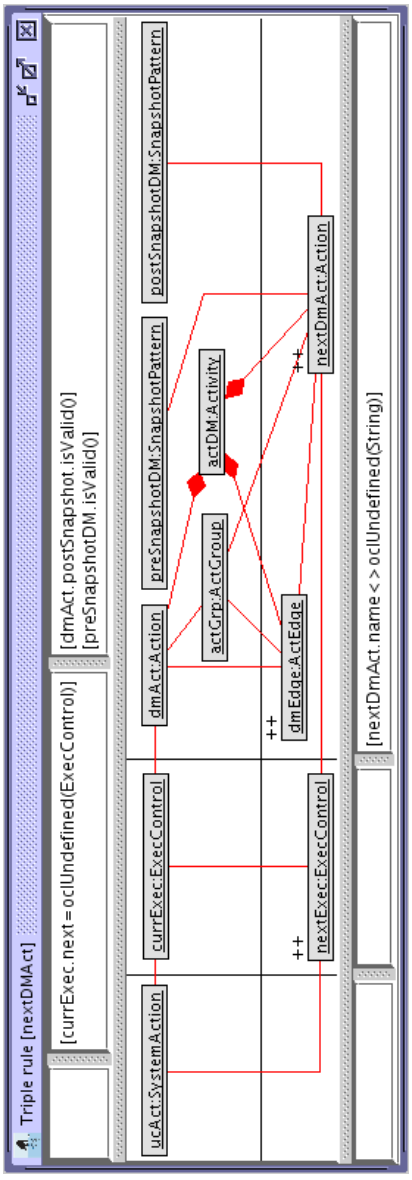


Figure 6.16: Triple rule for the next action at the design level

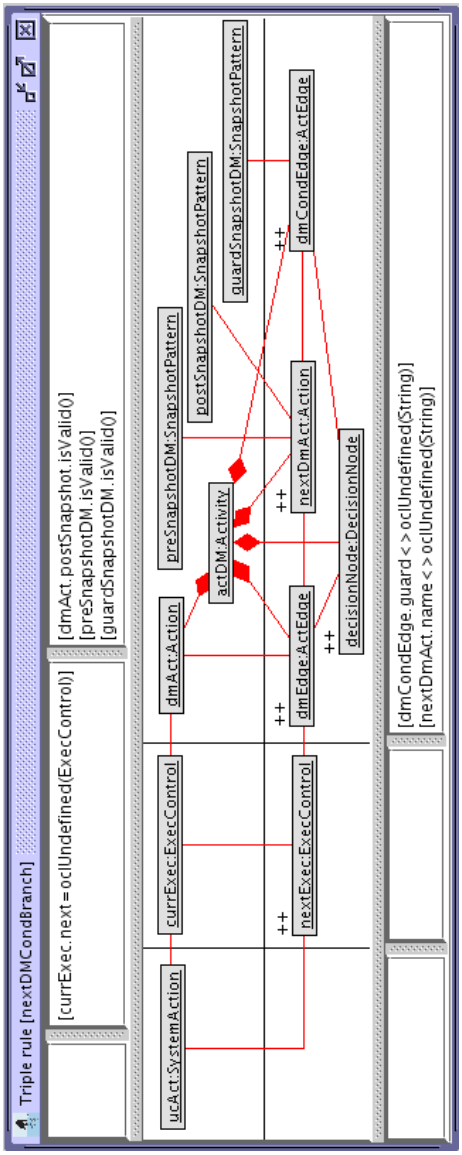


Figure 6.17: Triple rule for the next action with guard conditions at the design level

the current action at the design level to the next action in the same action group. This transfer needs to be checked by a guard condition. Like the `R5` rule, there is no transfer at the use case level in this execution step.

Example. The transfer by this rule is similar to the transfer from the action (5) to the action (6) as shown in Fig. 6.8.

R7. Triple rule for the next action in a new action group. The rule depicted in Fig. 6.18 is applied when the next execution step at the design level is the first action of a new action group. There is no transfer at the use case level in this execution step since these actions are part of an action sequence which realizes the corresponding system action at the use case level.

Example. This rule is applied for the transfer from the action (7) to the action (8) as presented in Fig. 6.8. The current action at the use case level is the action (4) as shown in Fig. 6.5.

R8. Triple rule for the next action in a new action group with guard conditions. The rule pictured in Fig. 6.19 is applied when the next execution step at the design level is the action of a new action group. There is a guard condition in this transfer. Like the `R7` rule, there is no transfer at the use case level in this execution step.

Example. This rule is applied for the transfer from the action (5) to the action (6) as presented in Fig. 6.8. The current action at the use case level is the action (4) as presented in Fig. 6.5.

R9. Triple rule for the next action in a use case extension. When the current action at the use case level is a system action, and the extension condition of this action is fulfilled, the next execution step will invoke the scenario of the extension use case. The corresponding execution step at the design level is a transfer from the current action to a new action group which realizes the extension scenario. This transfer needs to be checked by a guard condition. We can use the rule depicted in Fig. 6.20 to express this execution step.

The condition of an extension point is represented by a snapshot (the `extCondSnapshot` object) in the use case part of this rule. When this rule is applied, i.e., this condition is fulfilled, an `ExtensionPoint` object is created. A corresponding action group (the `actGrp` object) in the design part is created. The correspondence between them is represented by links to the `ExtPoint2ActGrp` object.

This rule also shows that the extension condition of a system action at the use case level is realized by a guard condition (the `guardSnapshotDM` object)

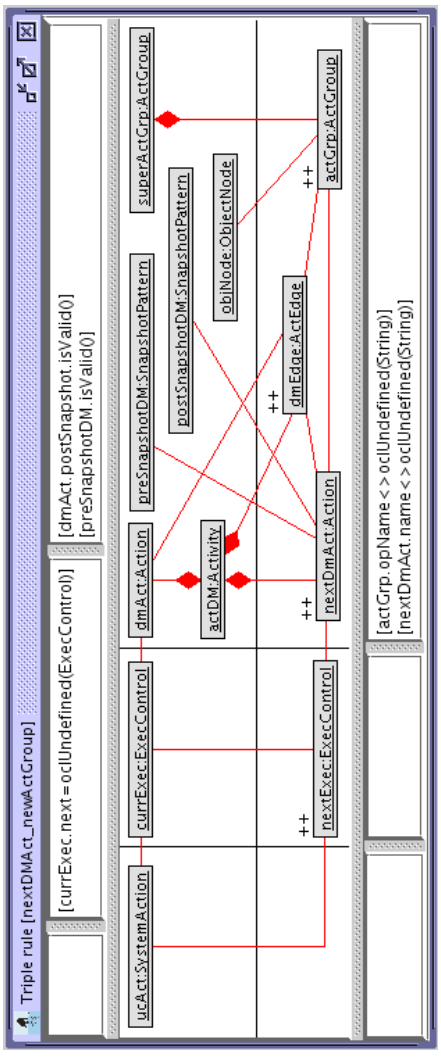


Figure 6.18: Triple rule for the next action in a new action group

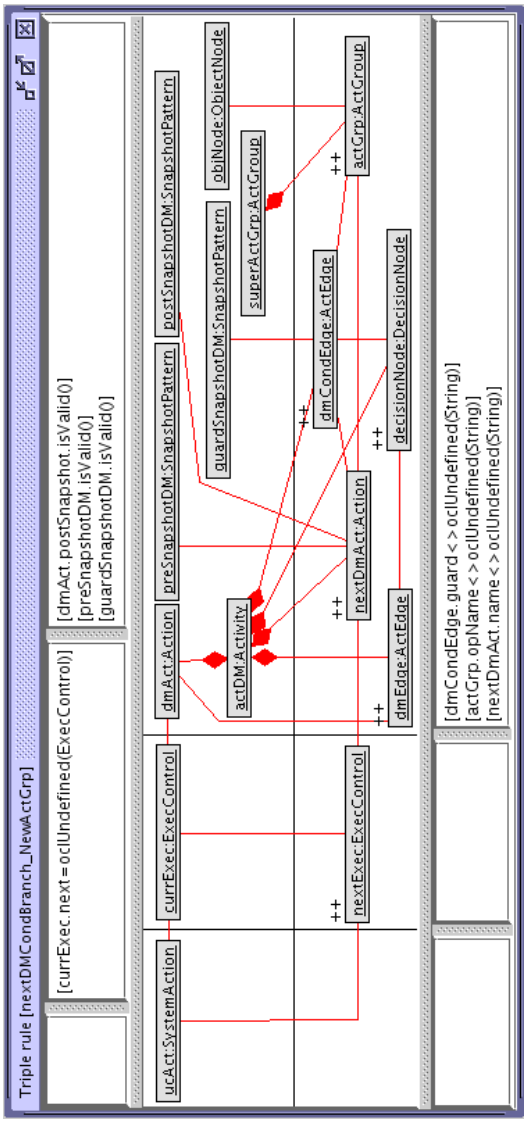


Figure 6.19: Triple rule for the next action in a new action group with guard conditions

together with a decision node (the `decisionNode` object) at the design level.

OCL conditions in the LHS of this rule allow us to validate snapshots, and to check whether the respective execution step is the current execution step. OCL conditions in the RHS allow checking the attribute values.

Example. Figure 6.5 shows an extension at the system action (4). The invocation of this extension point corresponds to the transfer from the action (5) to the action (10) as presented in Fig. 6.8. We can use this rule to express this execution step.

R10. Triple rule for the next use case action. We consider a situation in which the next execution step at the use case level is a use case action, i.e., at this step this use case includes another use case. The corresponding execution at the design level is a transfer from the current action to a new group action which represents the inclusion scenario. We can use the rule depicted in Fig. 6.21 to express this execution step.

When this rule is applied, a use case action (the `includedUcAct` object) is created in the use case part. A corresponding action group (the `actGrp`) is created in the design part. The correspondence between these actions is represented by links that are connected to the `UcAct2ActGrp` object.

OCL conditions in the LHS of this rule allow us to validate snapshots, and to check whether the respective execution step is the current execution step. OCL conditions in the RHS allow checking the attribute values.

Example. This rule is applied for the transfer from the action (4) to the action (5) as presented in Fig. 6.5. The corresponding execution at the design level is the transfer from the action (8) to the action (9) as shown in Fig. 6.8.

R11. Triple rule to finish the scenario. The rule presented in Fig. 6.22 is used to finish the execution of scenarios at the use case and design levels.

The postcondition of the scenario at the use case level is represented by a guard condition (the `guardSnapshotUC` object). This condition is checked before this rule is applied.

Once this rule is successfully applied we can conclude that the pair of scenarios at these use case and design levels in this execution are defined and there is no inconsistency between them.

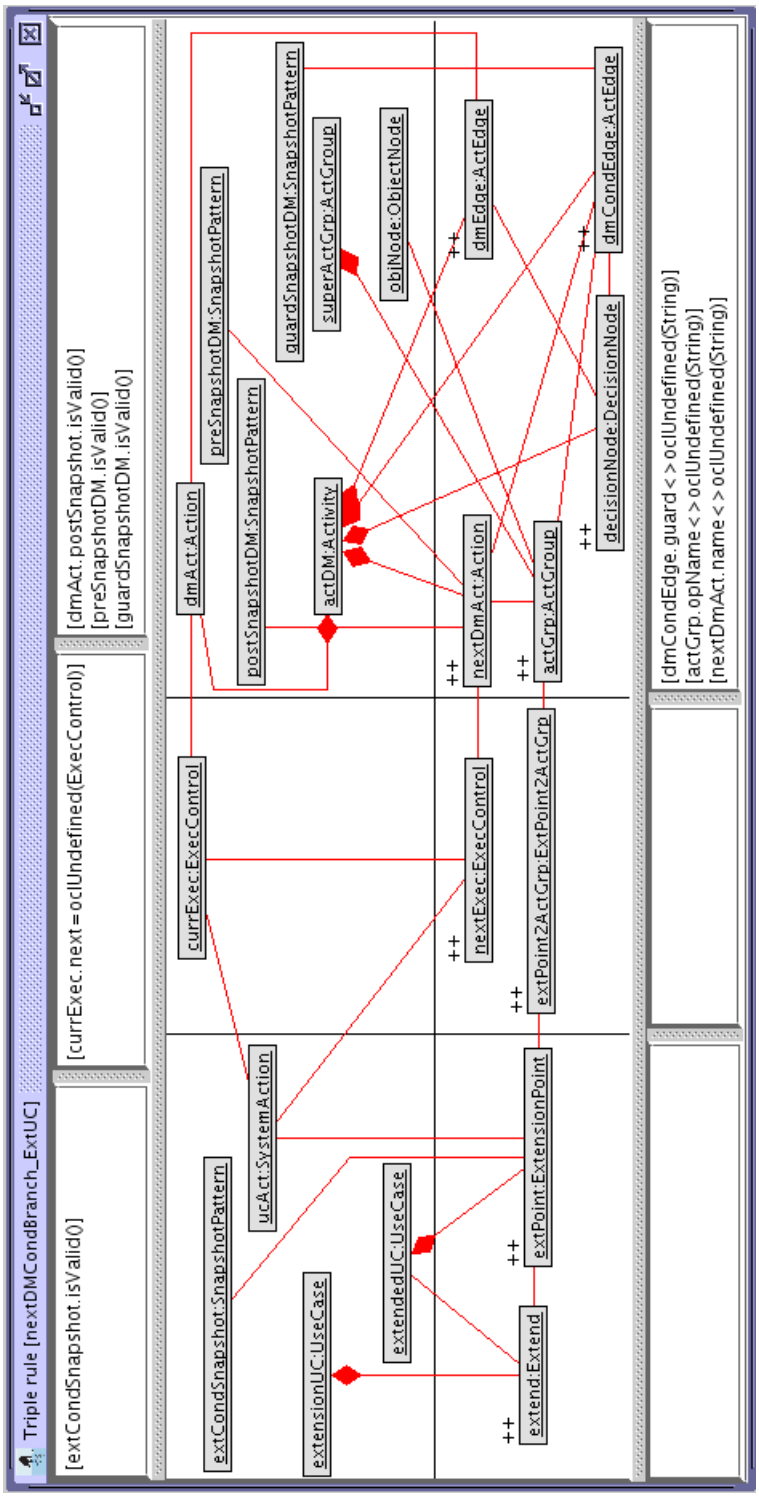


Figure 6.20: Triple rule for the next action in a use case extension

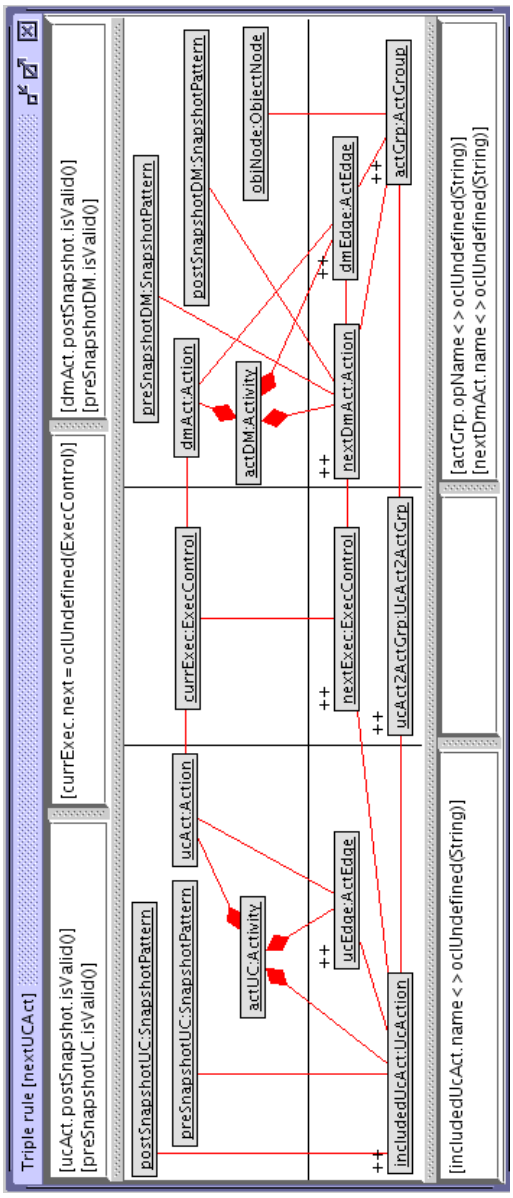


Figure 6.21: Triple rule for the next use case action

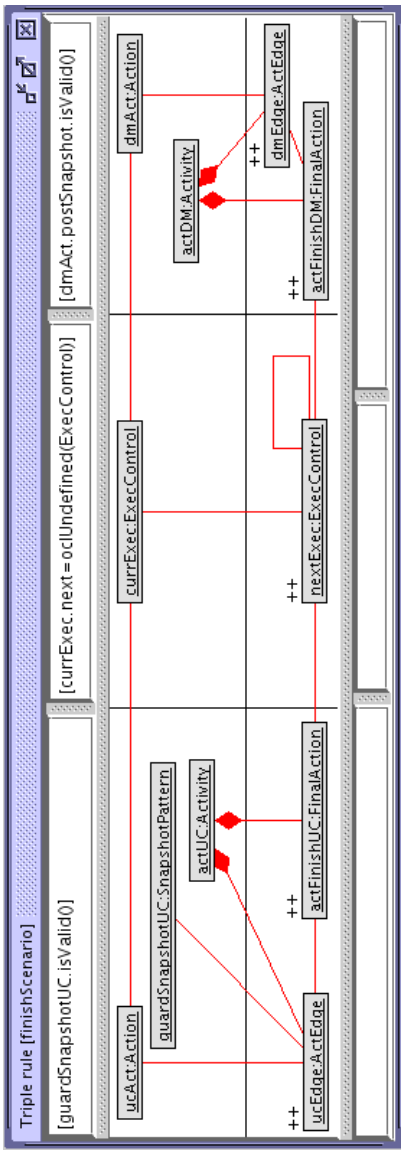


Figure 6.22: Triple rule to finish the scenario

```
rule nextSysAct
checkSource(
  ucAct:Action
  actUC:Activity
  preSnapshotUC:SnapshotPattern
  postSnapshotUC:SnapshotPattern
  (actUC,ucAct):ContainsNode
  [ucAct.postSnapshot.isValid()]
){
  sysAct:SystemAction
  ucEdge:ActEdge
  (actUC,sysAct):ContainsNode
  (actUC,ucEdge):ContainsEdge
  (ucEdge,sysAct):ConnectsTo
  (ucEdge,ucAct):ConnectsFrom
  (sysAct,preSnapshotUC):Pre_SnapshotPattern
  (sysAct,postSnapshotUC):Post_SnapshotPattern
  [sysAct.name<>oclUndefined(String)]}
checkTarget(
  dmAct:Action
  actDM:Activity
  objNode:ObjectNode
  preSnapshotDM:SnapshotPattern
  postSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
  [preSnapshotDM.isValid()]
){
  nextDmAct:Action
  dmEdge:ActEdge
  actGrp:ActGroup
  (actDM,nextDmAct):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,nextDmAct):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (actGrp,objNode):ActGroup_Object
  (actGrp,nextDmAct):GroupsNode
  (actGrp,dmEdge):GroupsEdge
  (nextDmAct,preSnapshotDM):Pre_SnapshotPattern
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [actGrp.opName<>oclUndefined(String)]
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  (ucAct,dmAct)as(ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)sysAct,nextDmAct) in nextExec:ExecControl
  (sysAct,actGrp) as (sysAct,actGrp)
                     in sysAct2ActGrp:SysAct2ActGrp
  (nextExec,currExec):NextExec}
end
```

Figure 6.23: USE4TGG description of the triple rule `R4` for the next system action

6.5 Synchronizing Scenarios

This section focuses on how a system execution can be represented by scenarios, and how we can synchronize them for the execution. We employ TGGs incorporating OCL in order to establish such a mechanism. The input of this mechanism is a set of execution scenarios described by use case models and design models. The output of this mechanism is a pair of scenarios for a system execution.

6.5.1 Applying Triple Rules Incorporating OCL

We employ derived triple rules for model integration in order to define pairs of scenarios at the use case and design levels for a system execution. For example, we consider a possible pair of scenarios for a system execution from the use case and design models shown in Fig. 6.5 and Fig. 6.8. The scenario at the use case level is the action sequence from the action (1) to the action (5) as pictured in Fig. 6.5. The extension point at the action (4) is not invoked in this scenario. The corresponding scenario at the design level is the action sequence from the action (1) to the action (9) as presented in Fig. 6.8. The co-evolution of snapshots in this case is carried out by triple rules in the rule sequence `[R1, R2, R4, R7, R3, R4, R8, R5, R7, R10, R11]`. We assume that the rules are applicable at corresponding system states.

We implement triple rules according to the explanation in Chapter 4 and Chapter 5. Triple rules are described in USE4TGG, and derived triple rules are realized as OCL operations in USE. For example, Fig. 6.23 depicts the USE4TGG description of the triple rule `R4` for the next system action.

6.5.2 Validating Snapshots

Before applying a triple rule we need to check the validation of the precondition snapshot of the current action and the postcondition snapshot of the previous action. In the implementation we use the `isValid()` operation of the `SnapshotPattern` class in order to check snapshots. This operation is realized (overridden) in concrete snapshot classes that inherit the `SnapshotPattern` class. A concrete snapshot is a snapshot as the pre- or postcondition of a concrete action (that inherits the `Action` class).

Example. Figure 6.24 shows the concrete classes `Pre_UpdateRental` and `Post_UpdateRental` corresponding to the pre- and postcondition snapshots

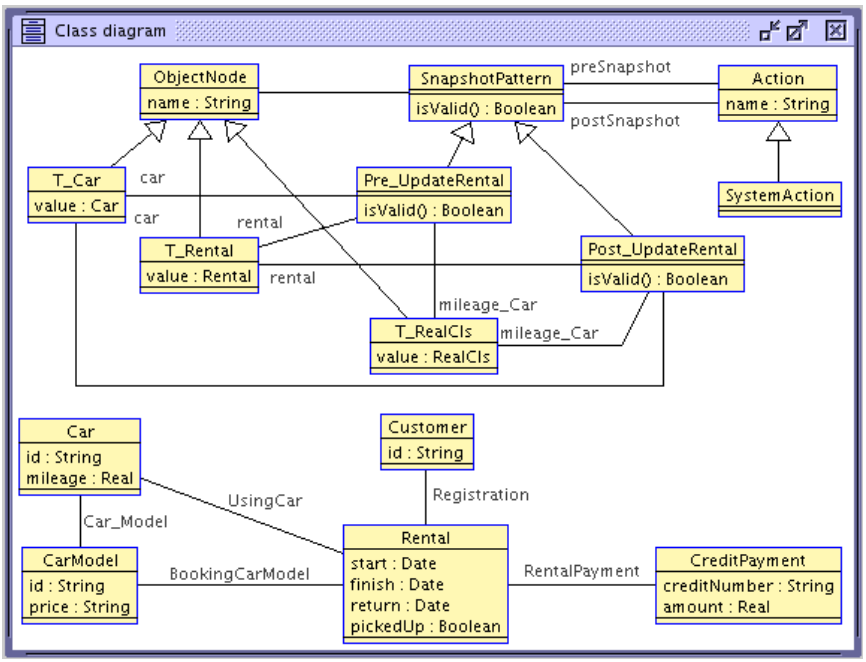


Figure 6.24: Representing snapshots in OCL

of the concrete action `UpdateRental`.

We can define the `isValid()` operation of a concrete snapshot class by an OCL boolean expression. This OCL boolean expression re-expresses in OCL the information of objects and links of this concrete snapshot. This translation can be illustrated with the translation in Fig. 5.1. Note that objects in concrete snapshots, i.e., the `ObjectNode` objects, play the role as object variables. They are instances of so-called object-pattern classes whose `value` attribute is an instance of the corresponding domain class. An object-pattern class, e.g., the `T_Car` class as shown in Fig. 6.24, can be recognized when its name is the concatenation of the string "T_" and the name of the corresponding domain class, the `Car` class in this case.

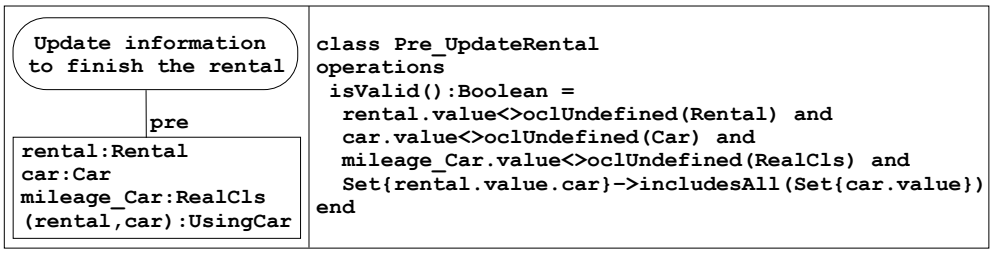


Figure 6.25: Translating a snapshot to an OCL boolean expression

A concrete snapshot accesses its concrete object using the role name (coin-

ciding to the object name in this snapshot) and the `value` attribute from the object-pattern class. For example, in Fig. 6.24 the `Pre_UpdateRental` snapshot accesses its concrete object, the `Car` object using the role name '`car`' and the `value` attribute from the `T_Car` class. Figure 6.25 shows an example for the translation of a concrete snapshot to an OCL boolean expression.

6.5.3 Performing the Transfer of Snapshots

Triple rules allow us to define next actions of a system execution. We need to perform the selected action after each rule application. Snapshots are defined depending on the kind of actions. Snapshots of the system action at the use case level are defined by snapshots of actor actions and system actions at the design level. Snapshots of actor actions are defined when the input from the actor is assigned by values. This process is manually performed in our implementation. Snapshots of system actions at the design level can be automatically defined by transformations as follows.

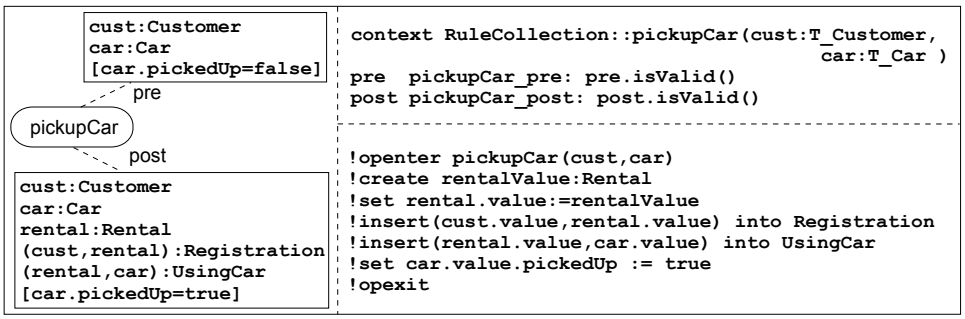


Figure 6.26: OCL operation to perform the `pickupCar` action

The pre- and postcondition snapshots of a system action at the design level can be seen as graphs with objects and links. This has been discussed in Sect. 5.2. Therefore, the performance of this action can be expressed as a graph transformation, and this transformation is realized by an OCL operation. Figure 6.26 shows an example for this transformation.

6.6 USE-based Implementation

This section overviews our implementation based on USE [GBR07] for describing the semantics of use cases on the basic of TGGs and OCL. Figure 6.27 shows the process model to be applied in this implementation.

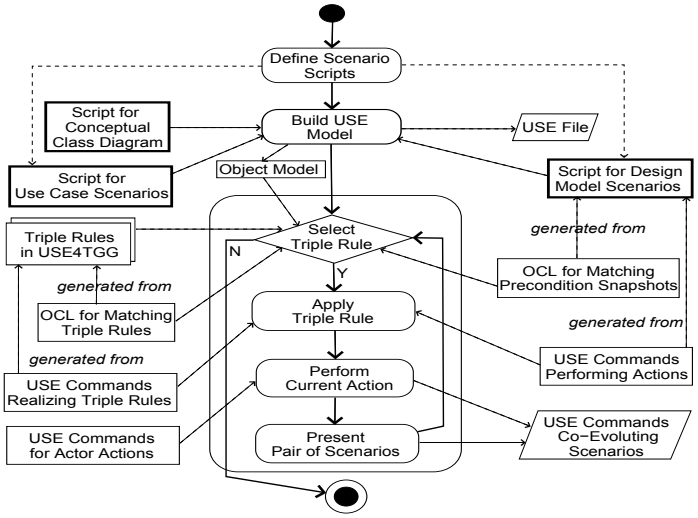


Figure 6.27: Process model for use case semantics

```
usecase carRental
action init startCarRental end
action final finishCarRental end

action enterCustInfo
post
 id_Cust:StringCls
 [id_Cust.string<>
          oclUndefined(String)]
end

action retrieveRentalInfo end

action updateCarInfo
post
 mileage_Car:RealCls
 [mileage_Car.real<>
             oclUndefined(Real)]

end

action updateRental
pre
 rental:Rental
 car:Car
 mileage_Car:RealCls
 (rental,car):UsingCar
post
 rental:Rental
 car:Car
 mileage_Car:RealCls
 (rental,car):UsingCar
 [car.mileage=mileage_Car.real]
ext returnLate
 rental:Rental
 today:Date
 [rental.finish<today.date]
end
```

```
action includeProcPayment
useCaseAction processPayment
end

transition
startCarRental->enterCustInfo
end

transition
enterCustInfo->retrieveRentalInfo
end

transition
retrieveRentalInfo->enterCustInfo
guard[else]
end

transition
retrieveRentalInfo->updateCarInfo
guard[T]
 cust:Customer
 rental:Rental
 id_Cust:StringCls
 (cust,rental):Registration
 [cust.id=id_Cust.string]
end

transition
updateCarInfo->updateRental
end

transition
updateRental->includeProcPayment
end

transition
includeProcPayment->finishCarRental
end
```

Figure 6.28: Use case specification in concrete syntax

The first step in this process model takes scripts for scenarios at the use case and design levels as input. The scripts are written in concrete textual syntax. Figure 6.28 shows the description in concrete textual syntax for the `Return Car` use case. The concrete textual syntax of scripts for design models is similar to the syntax for use case models. These scripts together with the USE script for the conceptual diagram are taken as the input for the second step in this process model. The output of this step is the USE file, which allows us to present system states and scenarios at the two levels.

The third step in this process model is the first step of four steps for a co-evolution step of snapshots at the use case and design levels for a system execution. This step aims to select the next triple rule. A triple rule is chosen by an OCL query, which is generated from the USE4TGG description of this rule. Before rules are matched, the precondition snapshot of actions at the design level need to be defined by OCL queries, which are generated from the script of design scenarios. The chosen triple rule is applied at the fourth step of this process model. The rule application is realized by USE commands, which are generated from the USE4TGG description of this rule. Figure 6.23 depicts the USE4TGG description of the `nextSysAct` triple rule.

When the chosen triple rule is applied, the current action is carried out. In case the current action is an actor action, it is performed by USE commands as the input of this process model. In case the current action is a system action at the design level, this action is performed by a transformation operation. This operation is realized by USE commands, which are generated from the script for design scenarios. At the final step of this process model the current state of scenarios at the use case and design levels is presented.

6.7 Related Work

Many researches as surveyed in [Hur97] have worked towards the introduction of rigor into use case descriptions. The work in [RB03, RAB96] proposes viewing use cases through the different levels of abstraction: The environment level states the relationship between use cases and services, the structure level views use cases as set of scenarios, and the event level specifies scenarios as a sequence of events. Similarly, in [Whi06] levels for modeling use cases include use case charts, scenario charts, and UML interaction diagrams. Within our approach we use the design model as another view of use cases in order to describe the semantics of use cases. But unlike those works, we offer an operational specification of use cases. We use triple rules incorporating

OCL in order to connect the use case and design levels. The sequence of rule applications represents the sequence of execution traces as well as the correspondence between them at these two levels.

The work in [SBN+07, DBGP04, RB03] employs the metamodel approach in order to form a conceptual frame for use case modeling. Those works only focus on the syntax aspect of use cases, whereas the formal semantics of use cases is the focus of our approach.

For a formal semantics of use cases, most of the works are strongly influenced by UML. They often establish the use case semantics based on activity diagrams or state charts. In fact, activity diagrams have been used in a variety of ways to support use-case based development [AJI04, LS06, GNE+08]. The work in [Whi06] proposes use case charts as an extension of activity diagrams for a trace-based semantics of use cases. The work in [SPW07, NFTJ06, PM03, GLST01] proposes using state charts to specify use cases. They aim to generate test cases from the use case specification. Those works consider use case scenarios as event sequences. The context of use cases which is mentioned in use case descriptions and determined by the conceptual domain is often ignored in those works. In our approach we combine the conceptual domain and extended activity diagrams in order to define use case scenarios.

The work in [JLMT08, HHT02, HH01] proposes using graph transformation and activity diagrams to specify use cases. Actions are specified by graph transformation rules. The LHS and RHS of the rule are pre- and postconditions (in a form like object diagrams) of the corresponding action. A scenario is represented as a rule sequence. Those works employ the technique analyzing a critical pair of rule sequences in order to check the dependence between use case scenarios. However, the semantics of use cases is not defined within that approach since scenarios at the use case level are undetermined. Our work for design scenarios as explained in Subsect. 6.5.3 is similar to that work. The difference is that contracts in our approach can be expressed with OCL conditions. Our aim is to increase the expressiveness of contracts.

This chapter continues our proposal for the approach to use cases in [Dan08, Dan07]. The core of this approach is to view use cases as a sequence of use case snapshots and to use the integration of TGGs and OCL to define this sequence. The integration of TGGs and OCL is the focus of Chapter 4, Chapter 5, and our previous work in [DG09a, GBD08].

Summary

This chapter has presented a case study showing the practical applicability of our model-driven approach. We propose using TGGs and OCL in order to describe the operational semantics of use cases. This approach allows us to advance use case models into Model-Driven Engineering. It is also a novel approach for describing the operational semantics of modeling languages.

Specifically, this chapter has presented triple rules incorporating OCL for describing use case semantics. They allow us to carry out the co-evolution of snapshots at the use case and design levels. In order to present snapshots, we have defined the use case metamodel and the metamodel based on the UML metamodel for scenarios at each level.

The mechanism to define scenarios in this chapter opens the possibility to simulate the automatic evolution of a system so that properties and non-functional requirements of the system can be checked. On the other hand, this can be seen as an effort to tackle the gap between use case and design models and to check the conformance between them. This results in basic constructions of an automatic and semi-automatic design.

Our approach is implemented in the USE tool, which supports full OCL. OCL plays an important role in our approach. It allows us to restrict triple rules and contracts, to validate snapshots, to perform actions as transformations, and to operationalize triple rules.

Chapter 7

Conclusion

The MDE paradigm puts forward a necessity as well as a challenge of a formal foundation for presenting precisely models and supporting automatic model manipulations. This thesis has contributed to defining such a formal foundation, developing support tools, and showing the practical applicability of the approach. In this chapter, we emphasize the key contributions of the thesis and discuss future work.

7.1 Key Contributions

This thesis makes three major contributions. The first major contribution is a formal foundation based on the incorporation of the object-oriented paradigm and triple graph transformation for a model-driven approach. For this foundation, TGGs have extended in several ways. The first extension is the integration of TGGs and OCL as explained in Chapter 4. That chapter also explains how application conditions of operations derived from triple rules including OCL can be defined. The second extension of TGGs is that triple rules can be deleting rules and the source and target sides of a triple rule can be connected by multiple links as presented in Chapter 3. The third extension is that the derived scenario of triple rules for model synchronization is proposed as explained in Chapter 5.

The second major contribution of the thesis is a USE-based tool for model transformation with an OCL-based transformation assurance frame. In order to present the integration of TGGs and OCL the USE4TGG language has been proposed as explained in Chapter 4. This language offers a declarative description of transformations as well as the correspondence between models. It can be compared to the QVT Relational language. In order to operationalize the integration of OCL and TGGs, an OCL-base approach has been proposed as explained in Chapter 5. This results in an OCL-based framework for model transformation. This framework can be compared to the QVT Operational Mapping language. Further, it offers means for trans-

formation quality assurance. This approach is realized based on the USE tool, which offers full OCL support.

The third major contribution of the thesis is showing the practical applicability of the model-driven approach based on the integration of OCL and TGGs. This is carried out with case studies of model transformation. They include (1) the transformation from a UML subset to a Java subset as explained in Chapter 5, (2) the transformation from UML activity diagrams to Communicating Sequential Processes (CSPs) as mentioned in [DGB07, VAB+08], (3) the transformation between logic languages as mentioned in Chapter 3, (4) the transformation between statecharts and extended hierarchical automata as presented in [DG09b], and (5) the relation between use case models and design models for an operation semantics of use cases as explained in Chapter 6. Especially, our approach for describing the operation semantics of use cases on the one hand can be broaden in order to describe the operational semantics of modeling languages. On the other hand, it allows us to generate scenarios as test cases, to validate system behavior, and to check the conformance between use case models and design models. This supports basic constructions of an automatic and semi-automatic design.

7.2 Future Work

The model-driven approach together with engineering solutions proposed in this thesis has contributed to the effort realizing the MDE vision. Substantial work remains toward the goal of achieving a full support for the MDE paradigm. In what follows we discuss several directions for future work.

7.2.1 Transformation, Consistency, and Traceability

The approach based on TGGs incorporating OCL for model transformation in this thesis allows us to detect and fix the inconsistency between models in a semi-automatic way. Our concern in future is how to generate OCL queries from triple rules incorporating OCL in order to detect such an inconsistency in an automatic way. We will explore different situations of the change of source and target models and develop algorithms to synchronize them. The challenge is how to locate such an inconsistency so that we can fix it by applying derived triple rules for model synchronization. Our future work also concerns the case of integrated models produced by deleting rules. We will develop a mechanism to keep the information of rule derivations, which

may be lost by deleting rules. It broadens our approach for maintaining the consistency between models.

The approach based on TGGs incorporating OCL in this thesis allows us to explain relationships between two models. In practice we often need to keep the consistency among many models. We plan to extend our approach for this aim. The case study for our work will be an extension of the case study of use cases presented in this thesis: Use case models and design models will be seen as aspect models. The focus of our work will be the concern how aspect models to be executed in synchronization.

We will evaluate the practical applicability of our OCL-based framework for model transformation with middle and large scale case studies. In particular, we see applications in the fundamentals of language engineering where a source language (including syntax and semantics) is translated into a target language (also including syntax and semantics). Transforming Domain-Specific Languages (DSLs) into core modeling languages is also our aim in future. The work in this thesis and in [DG09c] explains how TGGs incorporating OCL can be employed in order to describe operational semantics of modeling languages. We aim to extend this approach by studying its applicability for various other modeling languages.

With transformations based on TGGs incorporating OCL, we can check OCL properties of the target model. We plan to explore the possibility to prove such a property based on the specification of transformations with triple rules incorporating OCL. Here, properties of models can be expressed in alternate verification calculus, e.g., with temporal logic.

We plan to enhance features of our USE-based tool for model transformation, including algorithms for applying triple rules and the compiler for the extended USE4TGG language. We also focus on the feature that allows us to present the result of transformations in both concrete and abstract syntax.

7.2.2 Expressiveness of the Transformation Language

The USE4TGG language in our approach is a transformation language. This language must be studied with various case studies and extended in several ways in order to increase its expressiveness within the MDE context. First, transformations can be seen as artifacts. They may have relations such as composition, separation, and inheritance of transformations. This transformation language needs to be extended in order to support such features. We aim to extend our language to special QVT features expressed in QVT by

keywords ‘check’, ‘enforce’, and ‘realize’. Second, this language needs to support control structures for sequences of complex transformations. In parallel with such extensions, a corresponding formal foundation must be defined. We will focus on a structure that has a higher level of abstraction than triple rules and is similar to transformation units for plain graph transformation. A further activity will concentrate on how to generate correspondence metamodels from TGG rules.

7.2.3 Use Case Modeling Language

We have introduced an approach to use cases. Our general goal is a modeling language for use cases. We will focus on how to generate test cases from models in this language. This allows us to simulate the automatic evolution of the system in order that we can check properties as well as non-functional requirements, e.g., performance or security features of the designed system.

We will study transformations from use case models to other models such as User Interface models, design models, simulation models, and test models. Our approach to use cases is an effort to represent the textual description of use cases in a precise way. In future we continue the effort to narrow the gap between use case models and textual descriptions of use cases. We will also consider the possibility of representing use case models in natural language, where the core is to translate OCL conditions to natural language. In this way a use case model is really a requirements model of a system.

7.2.4 Expressiveness of OCL

OCL is the central ingredient of our current approach as well as of the future work mentioned above. During the implementation of our future work the applicability of OCL will be explored. Our concern is how to increase the expressiveness of OCL for the applications. Note that some extensions for OCL have been introduced recently such as temporal OCL [ZG03] and imperative OCL [OMG07a].

Appendix A

Implementation of USE4TGG

This appendix chapter describes our implementation of the USE4TGG language. Section A.1 presents the concrete textual syntax of USE4TGG. This syntax is an extension of the syntax of USE specifications. Section A.2 explains the abstract syntax of USE4TGG towards an integration of USE4TGG specifications in USE.

A.1 Concrete Textual Syntax

This section presents the grammar for the USE4TGG language. We use a BNF-like notation with the following conventions. Terminal symbols are bold strings enclosed in single quotes. The remaining strings are nonterminals. The nonterminal of the first production is the start symbol of the grammar. Elements of a sequence are separated by blanks, alternative elements are separated by a "|" symbol. Optional elements are enclosed in brackets "[" and "]", elements that can occur zero or more times are enclosed in braces "{" and "}". Parentheses "(" and ")" are used to group elements. Nonterminal symbols of the form "xId" have the same definition as the nonterminal "id". The `oclExpression` definition for OCL expressions is referred to [Ric02].

tggRuleCollection	::=	'**transformation**' trafoId
		{tggRuleDefinition}
tggRuleDefinition	::=	**'rule'** ruleId
		[**'mode' 'EXPLICIT'**|**'NONDELETING'**]
		'checkSource' ruleDefinition
		'checkTarget' ruleDefinition
		'checkCorr' corrRuleDefinition
		'end'
ruleDefinition	::=	'(' patternDefinition ')'
		'**{**' patternDefinition '**}**'
patternDefinition	::=	{objectDefinition}
		{linkDefinition}
		{conditionDefinition}

```
objectDefinition      ::= objectId ':' classId
linkDefinition        ::= '(' objectId ',' objectId ')' ':' assocId
conditionDefinition   ::= '[' oclExpression ']'
corrRuleDefinition    ::= '(' corrPatternDefinition ')'
                          '{' corrPatternDefinition '}'
corrPatternDefinition ::= {objectDefinition}
                          {corrLinkDefinition}
                          {linkDefinition}
                          {conditionDefinition}
corrLinkDefinition    ::= '(' [ '(' classId ')' ] objectId ','
                          [ '(' classId ')' ] objectId ')'
                          [ 'as' '(' roleId ',' roleId ')' ]
                          'in' objectId ':' objectId
id                    ::= ('a'..'z' | 'A'..'Z' | '_') { 'a'..'z' | 'A'..'Z' | '_' | '0'..'9'}
```

A.2 Abstract Syntax

Figure A.1 pictures the abstract syntax of the USE4TGG language. The meta-concepts `TggRuleCollection`, `MTggRule`, `MRule`, `MTggPattern`, and `MCorrLink` are explained in Subsect. 4.5.2. The other meta-concepts including `MObject`, `MLink`, `MSystemState`, and `OCLExpression` are mentioned in Sect. 2.3. In our implementation these concepts belong to the abstract syntax of USE specifications [Ric02].

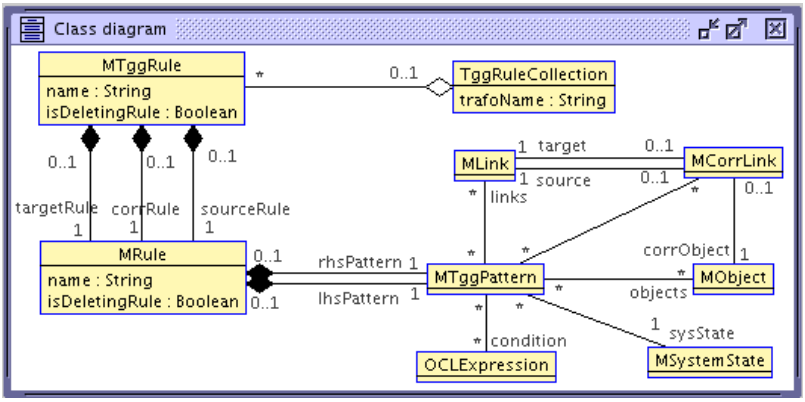


Figure A.1: Abstract syntax of the USE4TGG language

The USE4TGG abstract syntax allows us to integrate USE4TGG specifications in USE as illustrated in Fig A.2. This example triple rule is mentioned in our previous work [DG09b].

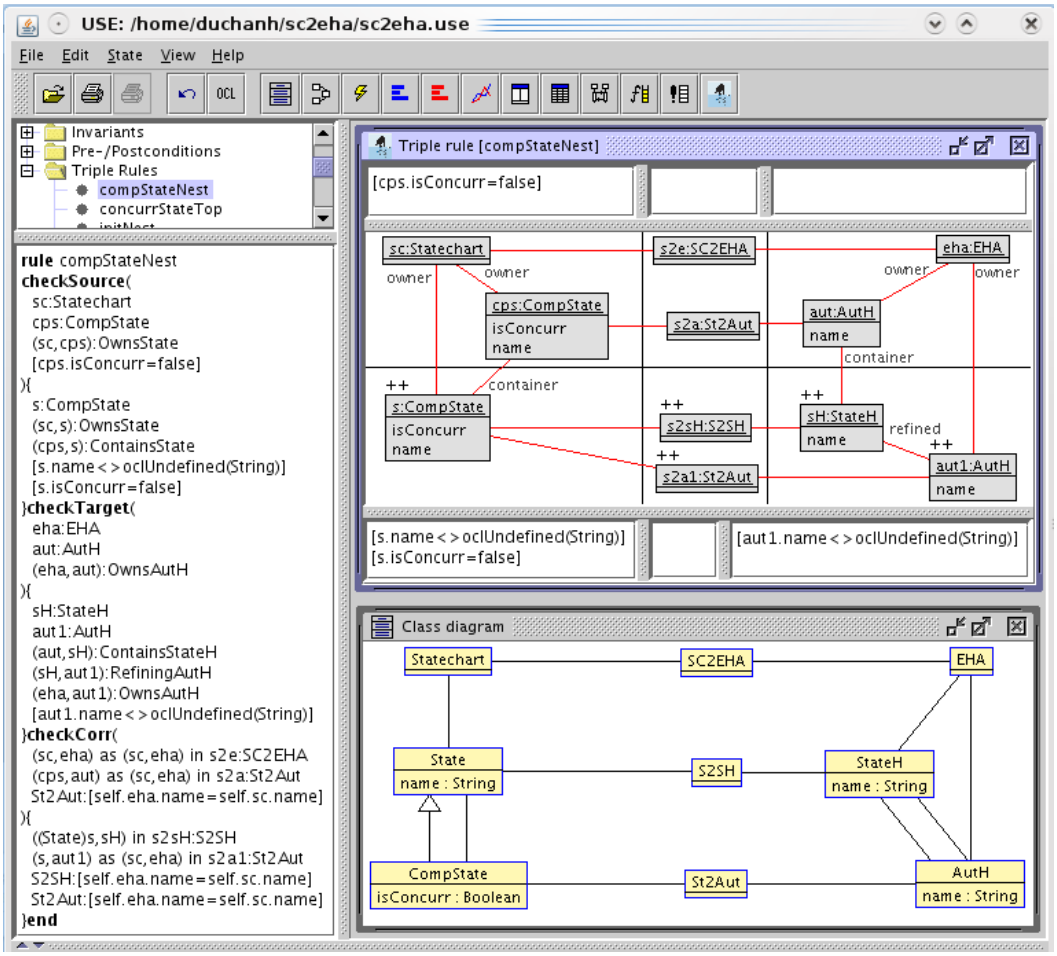


Figure A.2: Example triple rule in USE [DG09b]

Appendix B

Case Study of Use Case Semantics

This appendix section presents USE specifications and USE4TGG specifications for the case study of use case semantics as mentioned in Chapter 6. Specifically, Sect. B.1 contains the USE specification for the domain class diagram of the `CarRental` application. Section B.2 presents the USE specification of the structure of snapshots, that is generated from scripts of scenarios as explained in Sect. 6.6. Section B.3 lists the USE command sequence that illustrates how a pair of scenarios at the use case and design levels for an execution of the `ReturnCar` use case (i.e., a co-evolution of snapshots) is created in synchronization. A temporal state of this execution is depicted as in Sect. B.4. Section B.5 lists the USE4TGG specifications of rules for the UCM2DM transformation.

B.1 CarRental Model

```
model carRental_uc2dm
--------------------------------------------------------------------
-----------------------------------------CarRental Application
--------------------------------------------------------------------
class Customer
attributes
id: String
end
class Rental
attributes
 start: Date
 finish: Date
 return: Date
 pickedUp: Boolean
end
class Date
```

```
attributes
  value:Real
end
class CreditPayment
attributes
 creditNumber: String
 amount: Real
end
class CarModel
attributes
 id: String
 price: String
end
class Car
attributes
 id: String
 mileage: Real
end
-----------------------------------------Control Classes
class ReturningCar
end
class ReturnLate
end
class ProcessPayment
end
-----------------------------------------Associations
association Registration between
 Customer[1] role customer
 Rental[0..*] role rental
end
association BookingCarModel between
 Rental[0..*] role rental
 CarModel[1] role carModel
end
association UsingCar between
 Rental[0..*] role rental
 Car[0..1] role car
end
association RentalPayment between
 Rental[1] role rental
 CreditPayment[0..1] role payment
end
association Car_Model between
```

```
 CarModel[1] role carModel
 Car[0..*] role car
end
```

B.2 Structure of Snapshots

```
------------------------------------------------------------------
-------------------------------------------------Use Case Snapshot
------------------------------------------------------------------
class StringCls
attributes
 string:String
end
class RealCls
attributes
 real:Real
end
class DateCls
attributes
  date:Date
end
class T_Customer < ObjectNode
attributes
 value:Customer
end
class T_Rental < ObjectNode
attributes
 value:Rental
end
class T_Car < ObjectNode
attributes
 value:Car
end
class T_ReturningCar < ObjectNode
attributes
 value:ReturningCar
end
class T_ProcessPayment < ObjectNode
attributes
 value:ProcessPayment
end
class T_ReturnLate < ObjectNode
```

```
attributes
 value:ReturnLate
end
class T_StringCls < ObjectNode
attributes
 value:StringCls
end
class T_RealCls < ObjectNode
attributes
 value:RealCls
end
class T_DateCls < ObjectNode
attributes
 value:DateCls
end
------------------------------------------
class Post_EnterCustInfo < SnapshotPattern
operations
 isValid():Boolean = id_Cust.value<>oclUndefined(StringCls) and
  id_Cust.value.string<>oclUndefined(String)
end
association Post_EnterCustInfo_id_Cust between
 Post_EnterCustInfo[*] role post_EnterCustInfo_id_Cust
 T_StringCls[1] role id_Cust
end
------------------------------------------
class Guard_RetrieveRentalInfo_UpdateCarInfo < SnapshotPattern
operations
 isValid():Boolean = cust.value<>oclUndefined(Customer) and
  rental.value<>oclUndefined(Rental) and
  id_Cust.value<>oclUndefined(StringCls) and
  cust.value.rental->includes(rental.value) and
  cust.value.id = id_Cust.value.string
end
association Guard_RetrieveRentalInfo_UpdateCarInfo_cust between
 Guard_RetrieveRentalInfo_UpdateCarInfo[*]
                              role post_RetrieveRentalInfo_cust
 T_Customer[1] role cust
end
association Guard_RetrieveRentalInfo_UpdateCarInfo_rental between
 Guard_RetrieveRentalInfo_UpdateCarInfo[*]
                              role post_RetrieveRentalInfo_rental
 T_Rental[1] role rental
```

```
end
association Guard_RetrieveRentalInfo_UpdateCarInfo_id_Cust between
 Guard_RetrieveRentalInfo_UpdateCarInfo[*]
                                role post_RetrieveRentalInfo_id_Cust
 T_StringCls[1] role id_Cust
end
-----------------------------------------
class Post_UpdateCarInfo < SnapshotPattern
operations
 isValid():Boolean = mileage_Car.value<>oclUndefined(RealCls) and
  mileage_Car.value.real<>oclUndefined(Real)
end
association Post_UpdateCarInfo_mileage_Car between
 Post_UpdateCarInfo[*] role post_UpdateCarInfo_mileage_Car
 T_RealCls[1] role mileage_Car
end
-----------------------------------------
class Pre_UpdateRental < SnapshotPattern
operations
 isValid():Boolean = rental.value<>oclUndefined(Rental) and
  car.value<>oclUndefined(Car) and
  mileage_Car.value<>oclUndefined(RealCls) and
  Set{rental.value.car}->includesAll(Set{car.value})
end
association Pre_UpdateRental_rental between
 Pre_UpdateRental[*] role pre_UpdateRental_rental
 T_Rental[1] role rental
end
association Pre_UpdateRental_car between
 Pre_UpdateRental[*] role pre_UpdateRental_car
 T_Car[1] role car
end
association Pre_UpdateRental_mileage_Car between
 Pre_UpdateRental[*] role preSnapshot_mileage_Car
 T_RealCls[1] role mileage_Car
end
-----------------------------------------
class Post_UpdateRental < SnapshotPattern
operations
 isValid():Boolean = rental.value<>oclUndefined(Rental) and
  car.value<>oclUndefined(Car) and
  mileage_Car.value<>oclUndefined(RealCls) and
  Set{rental.value.car}->includesAll(Set{car.value}) and
```

```
  car.value.mileage = mileage_Car.value.real
end
association Post_UpdateRental_rental between
 Post_UpdateRental[*] role post_UpdateRental_rental
 T_Rental[1] role rental
end
association Post_UpdateRental_car between
 Post_UpdateRental[*] role post_UpdateRental_car
 T_Car[1] role car
end
association Post_UpdateRental_mileage_Car between
 Post_UpdateRental[*] role post_RetrieveRentalInfo_mileage_Car
 T_RealCls[1] role mileage_Car
end
-----------------------------------------
class ExtCond_UpdateRental < SnapshotPattern
operations
 isValid():Boolean = rental.value<>oclUndefined(Rental) and
  today.value<>oclUndefined(DateCls) and
  rental.value.finish.value < today.value.date.value
end
association ExtCond_UpdateRental_rental between
 ExtCond_UpdateRental[*] role extCond_UpdateRental_rental
 T_Rental[1] role rental
end
association ExtCond_UpdateRental_today between
 ExtCond_UpdateRental[*] role extCond_UpdateRental_today
 T_DateCls[1] role today
end
-------------------------------------------------------------------
--------------------------------------------------Design Snapshot
-------------------------------------------------------------------
class Pre_dmAct2 < SnapshotPattern
operations
 isValid():Boolean = cust.value<>oclUndefined(Customer) and
  id_Cust.value<>oclUndefined(StringCls) and
  cust.value.id = id_Cust.value.string
end
association Pre_dmAct2_cust between
 Pre_dmAct2[*] role preSnapshot_dmAct2_cust
 T_Customer[1] role cust
end
association Pre_dmAct2_id_Cust between
```

```
 Pre_dmAct2[*] role preSnapshot_dmAct2_id_Cust
 T_StringCls[1] role id_Cust
end
-----------------------------------------
class Pre_dmAct3 < SnapshotPattern
operations
 isValid():Boolean = cust.value<>oclUndefined(Customer) and
  rental.value<>oclUndefined(Rental) and
  cust.value.rental->includes(rental.value) and
  rental.value.return=oclUndefined(Date)
end
association Pre_dmAct3_cust between
 Pre_dmAct3[*] role preSnapshot_dmAct3_cust
 T_Customer[1] role cust
end
association Pre_dmAct3_rental between
 Pre_dmAct3[*] role preSnapshot_dmAct3_cust
 T_Rental[1] role rental
end
-----------------------------------------
class Pre_dmAct5 < SnapshotPattern
operations
 isValid():Boolean = today.value<>oclUndefined(DateCls)
end
association Pre_dmAct5_today between
 Pre_dmAct5[*] role preSnapshot_dmAct5_today
 T_DateCls[1] role today
end
-----------------------------------------
class Guard_dmAct5_dmAct6 < SnapshotPattern
operations
 isValid():Boolean = rental.value<>oclUndefined(Rental) and
  today.value<>oclUndefined(DateCls) and
  today.value.date.value<=rental.value.finish.value
end
association Guard_dmAct5_dmAct6_rental between
 Guard_dmAct5_dmAct6[*]
               role guard_dmAct5_dmAct6_rental
 T_Rental[1] role rental
end
association Guard_dmAct5_dmAct6_today between
 Guard_dmAct5_dmAct6[*]
               role guard_dmAct5_dmAct6_today
```

```
 T_DateCls[1] role today
end
-----------------------------------------
class Guard_dmAct5_dmAct10 < SnapshotPattern
operations
 isValid():Boolean=rental.value<>oclUndefined(Rental) and
  today.value<>oclUndefined(DateCls) and
  today.value.date.value> rental.value.finish.value
end
association Guard_dmAct5_dmAct10_rental between
 Guard_dmAct5_dmAct10[*]
               role guard_dmAct5_dmAct10_rental
 T_Rental[1] role rental
end
association Guard_dmAct5_dmAct10_today between
 Guard_dmAct5_dmAct10[*]
               role guard_dmAct5_dmAct10_today
 T_DateCls[1] role today
end
-----------------------------------------
class Pre_dmAct6 < SnapshotPattern
operations
 isValid():Boolean=rental.value<>oclUndefined(Rental) and
  today.value<>oclUndefined(DateCls)
end
association Pre_dmAct6_rental between
 Pre_dmAct6[*] role preSnapshot_dmAct6_rental
 T_Rental[1] role rental
end
association Pre_dmAct6_today between
 Pre_dmAct6[*] role preSnapshot_dmAct6_today
 T_DateCls[1] role today
end
-----------------------------------------
class Post_dmAct6 < SnapshotPattern
operations
 isValid():Boolean = rental.value<>oclUndefined(Rental) and
  today.value<>oclUndefined(DateCls) and
  rental.value.return.value=today.value.date.value
end
association Post_dmAct6_rental between
 Post_dmAct6[*] role postSnapshot_dmAct6_rental
 T_Rental[1] role rental
```

```
end
association Post_dmAct6_today between
 Post_dmAct6[*] role postSnapshot_dmAct6_today
 T_DateCls[1] role today
end
-----------------------------------------
class Pre_dmAct7 < SnapshotPattern
operations
 isValid():Boolean=rental.value<>oclUndefined(Rental) and
  car.value<>oclUndefined(Car) and
  Set{rental.value.car}->includesAll(Set{car.value})
end
association Pre_dmAct7_rental between
 Pre_dmAct7[*] role preSnapshot_dmAct7_rental
 T_Rental[1] role rental
end
association Pre_dmAct7_car between
 Pre_dmAct7[*] role preSnapshot_dmAct7_car
 T_Car[1] role car
end
-----------------------------------------
class Pre_dmAct8 < SnapshotPattern
operations
 isValid():Boolean = car.value<>oclUndefined(Car) and
  mileage_Car.value<>oclUndefined(RealCls)
end
association Pre_dmAct8_car between
 Pre_dmAct8[*] role preSnapshot_dmAct8_car
 T_Car[1] role car
end
association Pre_dmAct8_mileage_Car between
 Pre_dmAct8[*] role preSnapshot_dmAct8_mileage_Car
 T_RealCls[1] role mileage_Car
end
-----------------------------------------
class Post_dmAct8 < SnapshotPattern
operations
 isValid():Boolean = car.value<>oclUndefined(Car) and
  mileage_Car.value<>oclUndefined(RealCls) and
  car.value.mileage = mileage_Car.value.real
end
association Post_dmAct8_car between
 Post_dmAct8[*] role postSnapshot_dmAct8_car
```

```
 T_Car[1] role car
end
association Post_dmAct8_mileage_Car between
 Post_dmAct8[*] role postSnapshot_dmAct8_mileage_Car
 T_RealCls[1] role mileage_Car
end
```

B.3 Synchronization of Scenarios

```
-----------------------------------------------------------------------
----------------------------------------carRental_uc2dm_coEvol.cmd
-----------------------------------------------------------------------
open carRental_uc2dm_coEvol.use
!create rc: RuleCollection
!create uc:UseCase
!set uc.name := 'returnCar'
!create actUC:Activity
!create actDM:Activity
!insert (uc,actUC) into UC_Activity
!insert (uc,actDM) into UC_Activity
!create corrAct:Act2Act
!insert (corrAct,actUC) into L_Activity_Act2Act
!insert (corrAct,actDM) into R_Activity_Act2Act
!create defaultSnapshot:SnapshotPattern
-----------------------------------------------------------------------
-----------------------------------------------------CarRental INIT
-----------------------------------------------------------------------
!create currentDate:Date
!set currentDate.value:=20090105
!create bmw:Car
!set bmw.id:='HB01'
!create firstModel:CarModel
!insert (firstModel,bmw) into Car_Model
!create ada:Customer
!set ada.id:='ada'
!create adaRental:Rental
!insert (adaRental,firstModel) into BookingCarModel
!create startDate:Date
!set startDate.value:=20090101
!create finishDate:Date
!set finishDate.value:=20090115
!set adaRental.start:=startDate
```

```
!set adaRental.finish:=finishDate
!set adaRental.pickedUp:=true
!insert (ada,adaRental) into Registration
!insert (adaRental,bmw) into UsingCar
-----------------------------------------------------------------------
-----------------------------------------Begin Use Case ReturnCar
-----------------------------------------------------------------------
!let matchSL = Tuple{actUC:actUC,preSnapshotUC:defaultSnapshot}
!let matchTL = Tuple{actDM:actDM,preSnapshotDM:defaultSnapshot}
!let matchCL = Tuple{corrAct:corrAct}
read uc2dm_start_coEvol.cmd
-----------------------------------------------------------------------
-----------------------------------------------------NEXT_ActorAct1
-----------------------------------------------------------------------
-------------------------------------------------Post_EnterCustInfo
!create postSnapshot_EnterCustInfo:Post_EnterCustInfo
!create id_Cust:T_StringCls
!set id_Cust.name:='id_Cust'
!insert(postSnapshot_EnterCustInfo,id_Cust)
  into Post_EnterCustInfo_id_Cust
-----------------------------------------------------Match SysState
!create idCustStr:StringCls
!set idCustStr.string:='ada'
!set id_Cust.value:=idCustStr
-------------------------------------------------------------Execution
!let currExec=ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let _actorAct_name='RequireCarReturn'
!let _nextDmAct_name='getRentalInfo'
!let matchSL = Tuple{
  ucAct:ucAct,actUC:actUC,preSnapshotUC:defaultSnapshot,
  postSnapshotUC:postSnapshot_EnterCustInfo,
  _actorAct_name:_actorAct_name}
!let matchTL = Tuple{dmAct:dmAct,actDM:actDM,
  postSnapshotDM:defaultSnapshot,_nextDmAct_name:_nextDmAct_name}
!let matchCL = Tuple{currExec:currExec}
read uc2dm_nextActorAct_coEvol.cmd
-----------------------------------------------------------------------
-----------------------------------------------------NEXT_SysAct2
-----------------------------------------------------------------------
-----------------------------------------------Pre/Postcondition_UC
```

```
-------------------------------------------------Pre/Postcondition_DM
!create preSnapshot_dmAct2:Pre_dmAct2
!create cust:T_Customer
!set cust.name:='cust'
!insert (preSnapshot_dmAct2,cust) into Pre_dmAct2_cust
!insert (preSnapshot_dmAct2,id_Cust) into Pre_dmAct2_id_Cust
-----------------------------------------------------Match SysState
!set cust.value:=Customer.allInstances->any(id=id_Cust.value.string)
-------------------------------------------------------------Execution
!let currExec=ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!create retCar:T_ReturningCar
!let _actGrp_opName='getRentalInfo()'
!let _sysAct_name='RetrieveRentalInfo'
!let _nextDmAct_name='getCustomer'
!let matchSL=Tuple{
  ucAct:ucAct,actUC:actUC,preSnapshotUC:defaultSnapshot,
  postSnapshotUC:defaultSnapshot,_sysAct_name:_sysAct_name}
!let matchTL=Tuple{dmAct:dmAct,actDM:actDM,objNode:retCar,
  preSnapshotDM:preSnapshot_dmAct2,postSnapshotDM:preSnapshot_dmAct2,
  _actGrp_opName:_actGrp_opName,_nextDmAct_name:_nextDmAct_name}
!let matchCL=Tuple{currExec:currExec}
read uc2dm_nextSysAct_coEvol.cmd
-----------------------------------------------------------------------
-------------------------------------------------NEXT_DMAct3_newActGrp
-----------------------------------------------------------------------
-------------------------------------------------Pre/Postcondition_UC
-------------------------------------------------Pre/Postcondition_DM
!create preSnapshot_dmAct3:Pre_dmAct3
!create rental:T_Rental
!set rental.name:='rental'
!insert (preSnapshot_dmAct3,cust) into Pre_dmAct3_cust
!insert (preSnapshot_dmAct3,rental) into Pre_dmAct3_rental
-----------------------------------------------------Match SysState
!set rental.value:=cust.value.rental->any(return=oclUndefined(Date))
-------------------------------------------------------------Execution
!let currExec=ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let superActGrp = dmAct.actGrp->any(true)
```

```
!let _actGrp_opName='getRental()'
!let _nextDmAct_name='getRental'
!let matchSL=Tuple{ucAct:ucAct}
!let matchTL=Tuple{
  dmAct:dmAct,actDM:actDM,superActGrp:superActGrp,objNode:cust,
  preSnapshotDM:preSnapshot_dmAct3,postSnapshotDM:preSnapshot_dmAct3,
  _actGrp_opName:_actGrp_opName,_nextDmAct_name:_nextDmAct_name}
!let matchCL=Tuple{currExec:currExec}
read uc2dm_nextDMAct_newActGroup_coEvol.cmd
------------------------------------------------------------------
-----------------------------NEXT_ActorAct3_ConditionalBranch_coEvol
------------------------------------------------------------------
-------------------------------------------------Post_EnterCustInfo
!create postSnapshot_UpdateCarInfo:Post_UpdateCarInfo
!create mileage_Car:T_RealCls
!set mileage_Car.name:='mileage_Car'
!insert(postSnapshot_UpdateCarInfo,mileage_Car)
  into Post_UpdateCarInfo_mileage_Car
-------------------------------------
!create guard_RetrieveRentalInfo_UpdateCarInfo:
  Guard_RetrieveRentalInfo_UpdateCarInfo
!insert (guard_RetrieveRentalInfo_UpdateCarInfo,cust)
  into Guard_RetrieveRentalInfo_UpdateCarInfo_cust
!insert (guard_RetrieveRentalInfo_UpdateCarInfo,rental)
  into Guard_RetrieveRentalInfo_UpdateCarInfo_rental
!insert (guard_RetrieveRentalInfo_UpdateCarInfo,id_Cust)
  into Guard_RetrieveRentalInfo_UpdateCarInfo_id_Cust
-----------------------------------------------------Match SysState
!create mileageCarReal:RealCls
!set mileageCarReal.real:=10000
!set mileage_Car.value:=mileageCarReal
-------------------------------------------------------------Execution
!let currExec = ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let _actorAct_name='supplyRentalInfo'
!let _nextDmAct_name='updateRentalInfo'
!let matchSL = Tuple{ucAct:ucAct,actUC:actUC,
  preSnapshotUC:defaultSnapshot,
  postSnapshotUC:postSnapshot_UpdateCarInfo,
  guardSnapshotUC:guard_RetrieveRentalInfo_UpdateCarInfo,
  _actorAct_name:_actorAct_name}
```

```
!let matchTL = Tuple{dmAct:dmAct,actDM:actDM,
  postSnapshotDM:defaultSnapshot,_nextDmAct_name:_nextDmAct_name}
!let matchCL = Tuple{currExec:currExec}
read uc2dm_nextActorAct_ConditionalBranch_coEvol.cmd
------------------------------------------------------------------
-----------------------------------------------------NEXT_SysAct4
------------------------------------------------------------------
-----------------------------------------------Pre/Postcondition_UC
!create preSnapshot_UpdateRental:Pre_UpdateRental
!create car:T_Car
!set car.name:='car'
!insert(preSnapshot_UpdateRental,rental)
  into Pre_UpdateRental_rental
!insert(preSnapshot_UpdateRental,car)
  into Pre_UpdateRental_car
!insert(preSnapshot_UpdateRental,mileage_Car)
  into Pre_UpdateRental_mileage_Car
-----------------------------------------------
!create postSnapshot_UpdateRental:Post_UpdateRental
!insert(postSnapshot_UpdateRental,rental)
  into Post_UpdateRental_rental
!insert(postSnapshot_UpdateRental,car)
  into Post_UpdateRental_car
!insert(postSnapshot_UpdateRental,mileage_Car)
  into Post_UpdateRental_mileage_Car
-----------------------------------------------Pre/Postcondition_DM
!create preSnapshot_dmAct5:Pre_dmAct5
!create today:T_DateCls
!set today.name:='today'
!insert (preSnapshot_dmAct5,today) into Pre_dmAct5_today
-----------------------------------------------------Match SysState
!create todayDateCls:DateCls
!set todayDateCls.date:=currentDate
!set today.value:=todayDateCls
------------------------------------------------------------Execution
!let currExec = ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let _actGrp_opName='updateRentalInfo()'
!let _sysAct_name='UpdateRentalInfo'
!let _nextDmAct_name='getCurrDay'
!let matchSL=Tuple{ucAct:ucAct,
```

```
 actUC:actUC,preSnapshotUC:preSnapshot_UpdateRental,
 postSnapshotUC:postSnapshot_UpdateRental,_sysAct_name:_sysAct_name}
!let matchTL=Tuple{dmAct:dmAct,actDM:actDM,objNode:retCar,
 preSnapshotDM:preSnapshot_dmAct5,postSnapshotDM:preSnapshot_dmAct5,
 _actGrp_opName:_actGrp_opName,_nextDmAct_name:_nextDmAct_name}
!let matchCL=Tuple{currExec:currExec}
read uc2dm_nextSysAct_coEvol.cmd
-----------------------------------------------------------------
-------------------------------------NEXT_DMCondBranch56_NewActGrp
-----------------------------------------------------------------
-----------------------------------------------Pre/Postcondition_UC
-----------------------------------------------Pre/Postcondition_DM
!create preSnapshot_dmAct6:Pre_dmAct6
!insert (preSnapshot_dmAct6,rental) into Pre_dmAct6_rental
!insert (preSnapshot_dmAct6,today) into Pre_dmAct6_today
---------------------------------
!create postSnapshot_dmAct6:Post_dmAct6
!insert(postSnapshot_dmAct6,rental) into Post_dmAct6_rental
!insert (postSnapshot_dmAct6,today) into Post_dmAct6_today
---------------------------------
!create guard_dmAct5_dmAct6:Guard_dmAct5_dmAct6
!insert(guard_dmAct5_dmAct6,rental)
  into Guard_dmAct5_dmAct6_rental
!insert(guard_dmAct5_dmAct6,today)
  into Guard_dmAct5_dmAct6_today
-----------------------------------------------------Match SysState
-------------------------------------------------------------Execution
!let currExec = ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let superActGrp = dmAct.actGrp->any(true)
!let _dmCondEdge_guard='else'
!let _actGrp_opName='updateRental'
!let _nextDmAct_name='updateReturn'
!let matchSL=Tuple{ucAct:ucAct}
!let matchTL=Tuple{dmAct:dmAct,actDM:actDM,superActGrp:superActGrp,
  objNode:rental,preSnapshotDM:preSnapshot_dmAct6,
  postSnapshotDM:postSnapshot_dmAct6,
  guardSnapshotDM:guard_dmAct5_dmAct6,
  _dmCondEdge_guard:_dmCondEdge_guard,_actGrp_opName:_actGrp_opName,
  _nextDmAct_name:_nextDmAct_name}
!let matchCL=Tuple{currExec:currExec}
```

```
read uc2dm_nextDMCondBranch_NewActGrp_coEvol.cmd
-----------------------------------------------------Update SysState
!set rental.value.return:=today.value.date
-------------------------------------------------------------------
-----------------------------------------------------NEXT_DMAct7
-------------------------------------------------------------------
-----------------------------------------------Pre/Postcondition_UC
-----------------------------------------------Pre/Postcondition_DM
!create preSnapshot_dmAct7:Pre_dmAct7
!insert (preSnapshot_dmAct7,rental) into Pre_dmAct7_rental
!insert (preSnapshot_dmAct7,car) into Pre_dmAct7_car
-----------------------------------------------------Match SysState
!set car.value:=rental.value.car
-------------------------------------------------------------Execution
!let currExec = ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let actGrp = dmAct.actGrp->any(true)
!let _nextDmAct_name='getCar'
!let matchSL=Tuple{ucAct:ucAct}
!let matchTL=Tuple{dmAct:dmAct,actDM:actDM,actGrp:actGrp,
 preSnapshotDM:preSnapshot_dmAct7,postSnapshotDM:preSnapshot_dmAct7,
 _nextDmAct_name:_nextDmAct_name}
!let matchCL=Tuple{currExec:currExec}
read uc2dm_nextDMAct_coEvol.cmd
-------------------------------------------------------------------
-----------------------------------------NEXT_DMAct8_newActGroup
-------------------------------------------------------------------
-----------------------------------------------Pre/Postcondition_DM
!create preSnapshot_dmAct8:Pre_dmAct8
!insert (preSnapshot_dmAct8,car) into Pre_dmAct8_car
!insert (preSnapshot_dmAct8,mileage_Car)
  into Pre_dmAct8_mileage_Car
-----------------------------------
!create postSnapshot_dmAct8:Post_dmAct8
!insert (postSnapshot_dmAct8,car) into Post_dmAct8_car
!insert (postSnapshot_dmAct8,mileage_Car)
  into Post_dmAct8_mileage_Car
-----------------------------------------------------Match SysState
-------------------------------------------------------------Execution
!let currExec = ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
```

```
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let superActGrp = dmAct.actGrp->any(true).superGroup
!let _actGrp_opName='updateCar'
!let _nextDmAct_name='updateCar'
!let matchSL=Tuple{ucAct:ucAct}
!let matchTL=Tuple{dmAct:dmAct,actDM:actDM,superActGrp:superActGrp,
  objNode:car,preSnapshotDM:preSnapshot_dmAct8,
  postSnapshotDM:postSnapshot_dmAct8,_actGrp_opName:_actGrp_opName,
  _nextDmAct_name:_nextDmAct_name}
!let matchCL=Tuple{currExec:currExec}
read uc2dm_nextDMAct_newActGroup_coEvol.cmd
-----------------------------------------------------Update SysState
!set car.value.mileage:=mileage_Car.value.real
-------------------------------------------------------------------
-----------------------------------------------------NEXT_ucAct
-------------------------------------------------------------------
-----------------------------------------------Pre/Postcondition_UC
-----------------------------------------------Pre/Postcondition_DM
-----------------------------------------------------Match SysState
-------------------------------------------------------------Execution
!let currExec = ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let _includedUcAct_name='ProcessPayment'
!let _actGrp_opName = 'processPayment'
!let _nextDmAct_name='processPayment'
!create processPayment:T_ProcessPayment
!let matchSL=Tuple{ucAct:ucAct,actUC:actUC,
  preSnapshotUC:defaultSnapshot,postSnapshotUC:defaultSnapshot,
  _includedUcAct_name:_includedUcAct_name}
!let matchTL=Tuple{dmAct:dmAct,actDM:actDM,objNode:processPayment,
  preSnapshotDM:defaultSnapshot,postSnapshotDM:defaultSnapshot,
  _actGrp_opName:_actGrp_opName,_nextDmAct_name:_nextDmAct_name}
!let matchCL=Tuple{currExec:currExec}
read uc2dm_nextUCAct_coEvol.cmd
-------------------------------------------------------------------
-----------------------------------------------------Finish UC
-------------------------------------------------------------------
-----------------------------------------------Pre/Postcondition_UC
-----------------------------------------------Pre/Postcondition_DM
-----------------------------------------------------Match SysState
```

```
-------------------------------------------------------Execution
!let currExec = ExecControl.allInstances->any(
  next=oclUndefined(ExecControl))
!let ucAct = currExec.ucAct
!let dmAct = currExec.dmAct
!let matchSL=Tuple{ucAct:ucAct,actUC:actUC,
  guardSnapshotUC:defaultSnapshot}
!let matchTL=Tuple{dmAct:dmAct,actDM:actDM}
!let matchCL=Tuple{currExec:currExec}
read uc2dm_finish_coEvol.cmd
```

B.4 Example State

Figure B.1 depicts an example state of the synchronization of scenarios. The object diagram in the bottom part shows the current state of the system. The object diagram on top presents an integrated model of scenarios at the use case and design levels. This object diagram can be seen as a triple graph. Highlighted objects belong to the correspondence part of the triple graph. Scenarios at the use case and design levels correspond to the source (up) and target(bottom) parts of the triple graph, respectively.

B.5 UCM2DM Transformation Rules

```
-----------------------------------------------------------------
-----------------------------------------------------------RULES
-----------------------------------------------------------------
transformation uc2dm
rule startScenario
mode NONDELETING
checkSource(
  actUC:Activity
  preSnapshotUC:SnapshotPattern
  [preSnapshotUC.isValid()]
){
  actInitUC:InitialAction
  (actUC,actInitUC):ContainsNode
  (actInitUC,preSnapshotUC):Pre_SnapshotPattern
  (actInitUC,preSnapshotUC):Post_SnapshotPattern}
checkTarget(
  actDM:Activity
```

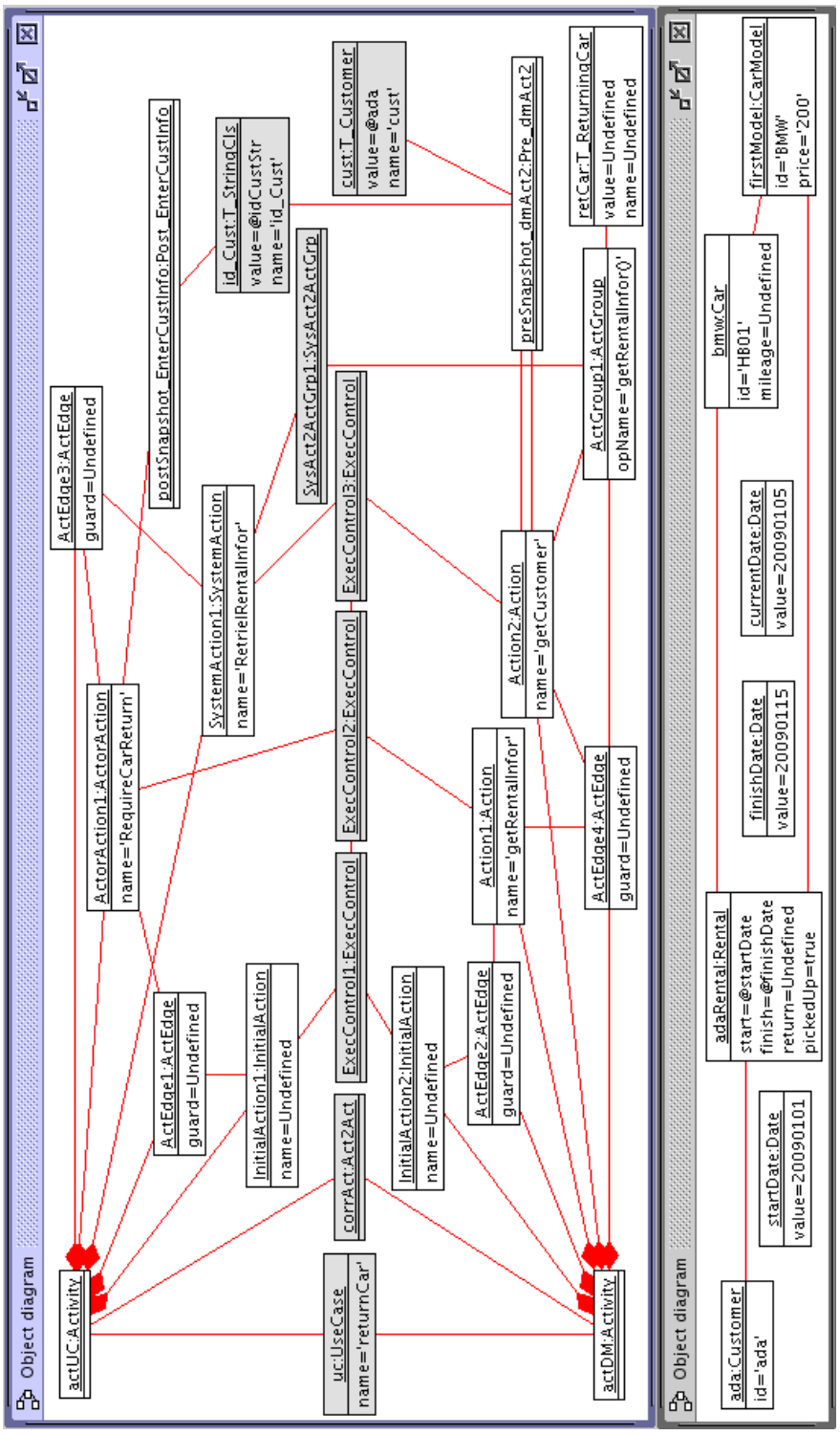


Figure B.1: State of the synchronization of scenarios

```
  preSnapshotDM:SnapshotPattern
  [preSnapshotDM.isValid()]
){
  actInitDM:InitialAction
  (actDM,actInitDM):ContainsNode
  (actInitDM,preSnapshotDM):Pre_SnapshotPattern
  (actInitDM,preSnapshotDM):Post_SnapshotPattern}
checkCorr(
  (actUC,actDM) as (usecase,design) in corrAct:Act2Act
  Act2Act:[self.usecase.uc = self.design.uc]
){
  ((Action)actInitUC,(Action)actInitDM)
    as (ucAct,dmAct) in currExec:ExecControl}
end
-----------------------------------------------------------------------
rule nextActorAct
checkSource(
  ucAct:Action
  actUC:Activity
  preSnapshotUC:SnapshotPattern
  postSnapshotUC:SnapshotPattern
  (actUC,ucAct):ContainsNode
  [ucAct.postSnapshot.isValid()]
  [preSnapshotUC.isValid()]
){
  actorAct:ActorAction
  ucEdge:ActEdge
  (actUC,actorAct):ContainsNode
  (actUC,ucEdge):ContainsEdge
  (ucEdge,actorAct):ConnectsTo
  (ucEdge,ucAct):ConnectsFrom
  (actorAct,preSnapshotUC):Pre_SnapshotPattern
  (actorAct,postSnapshotUC):Post_SnapshotPattern
  [actorAct.name<>oclUndefined(String)]}
checkTarget(
  dmAct:Action
  actDM:Activity
  postSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
){
  nextDmAct:Action
  dmEdge:ActEdge
```

```
  (actDM,nextDmAct):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,nextDmAct):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  (ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)actorAct,nextDmAct) in nextExec:ExecControl
  (nextExec,currExec):NextExec}
end
-----------------------------------------------------------------------
rule nextActorAct_ConditionalBranch
checkSource(
  ucAct:Action
  actUC:Activity
  preSnapshotUC:SnapshotPattern
  postSnapshotUC:SnapshotPattern
  guardSnapshotUC:SnapshotPattern
  (actUC,ucAct):ContainsNode
  [ucAct.postSnapshot.isValid()]
  [preSnapshotUC.isValid()]
  [guardSnapshotUC.isValid()]
){
  actorAct:ActorAction
  ucEdge:ActEdge
  (actUC,actorAct):ContainsNode
  (actUC,ucEdge):ContainsEdge
  (ucEdge,actorAct):ConnectsTo
  (ucEdge,ucAct):ConnectsFrom
  (ucEdge,guardSnapshotUC):GuardEdge
  (actorAct,preSnapshotUC):Pre_SnapshotPattern
  (actorAct,postSnapshotUC):Post_SnapshotPattern
  [actorAct.name<>oclUndefined(String)]}
checkTarget(
  dmAct:Action
  actDM:Activity
  postSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
){
```

```
  nextDmAct:Action
  dmEdge:ActEdge
  (actDM,nextDmAct):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,nextDmAct):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  (ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)actorAct,nextDmAct) in nextExec:ExecControl
  (nextExec,currExec):NextExec}
end
-----------------------------------------------------------------------
rule nextSysAct
checkSource(
  ucAct:Action
  actUC:Activity
  preSnapshotUC:SnapshotPattern
  postSnapshotUC:SnapshotPattern
  (actUC,ucAct):ContainsNode
  [ucAct.postSnapshot.isValid()]
  [preSnapshotUC.isValid()]
){
  sysAct:SystemAction
  ucEdge:ActEdge
  (actUC,sysAct):ContainsNode
  (actUC,ucEdge):ContainsEdge
  (ucEdge,sysAct):ConnectsTo
  (ucEdge,ucAct):ConnectsFrom
  (sysAct,preSnapshotUC):Pre_SnapshotPattern
  (sysAct,postSnapshotUC):Post_SnapshotPattern
  [sysAct.name<>oclUndefined(String)]}
checkTarget(
  dmAct:Action
  actDM:Activity
  objNode:ObjectNode
  preSnapshotDM:SnapshotPattern
  postSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
```

```
  [preSnapshotDM.isValid()]
){
  nextDmAct:Action
  dmEdge:ActEdge
  actGrp:ActGroup
  (actDM,nextDmAct):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,nextDmAct):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (actGrp,objNode):ActGroup_Object
  (actGrp,nextDmAct):GroupsNode
  (actGrp,dmEdge):GroupsEdge
  (nextDmAct,preSnapshotDM):Pre_SnapshotPattern
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [actGrp.opName<>oclUndefined(String)]
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  (ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)sysAct,nextDmAct) in nextExec:ExecControl
  (sysAct,actGrp) as (sysAct,actGrp) in sysAct2ActGrp:SysAct2ActGrp
  (nextExec,currExec):NextExec}
end
-----------------------------------------------------------------------
rule nextDMAct
checkSource(
  ucAct:SystemAction
){
}checkTarget(
  dmAct:Action
  actDM:Activity
  actGrp:ActGroup
  preSnapshotDM:SnapshotPattern
  postSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  (actGrp,dmAct):GroupsNode
  [dmAct.postSnapshot.isValid()]
  [preSnapshotDM.isValid()]
){
  nextDmAct:Action
  dmEdge:ActEdge
  (actDM,nextDmAct):ContainsNode
```

```
  (actDM,dmEdge):ContainsEdge
  (dmEdge,nextDmAct):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (actGrp,nextDmAct):GroupsNode
  (actGrp,dmEdge):GroupsEdge
  (nextDmAct,preSnapshotDM):Pre_SnapshotPattern
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  ((Action)ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)ucAct,nextDmAct) in nextExec:ExecControl
  (nextExec,currExec):NextExec}
end
--------------------------------------------------------------------
rule nextDMAct_newActGroup
checkSource(
  ucAct:SystemAction
){
}checkTarget(
  dmAct:Action
  actDM:Activity
  superActGrp:ActGroup
  objNode:ObjectNode
  preSnapshotDM:SnapshotPattern
  postSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
  [preSnapshotDM.isValid()]
){
  nextDmAct:Action
  dmEdge:ActEdge
  actGrp:ActGroup
  (actDM,nextDmAct):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,nextDmAct):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (actGrp,objNode):ActGroup_Object
  (actGrp,nextDmAct):GroupsNode
  (actGrp,dmEdge):GroupsEdge
  (superActGrp,actGrp):SubGroup
  (nextDmAct,preSnapshotDM):Pre_SnapshotPattern
```

```
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [actGrp.opName<>oclUndefined(String)]
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  ((Action)ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)ucAct,nextDmAct) in nextExec:ExecControl
  (nextExec,currExec):NextExec}
end
-----------------------------------------------------------------------
rule nextDMCondBranch
checkSource(
  ucAct:SystemAction
){
}checkTarget(
  dmAct:Action
  actDM:Activity
  preSnapshotDM:SnapshotPattern
  postSnapshotDM:SnapshotPattern
  guardSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
  [preSnapshotDM.isValid()]
  [guardSnapshotDM.isValid()]
){
  decisionNode:DecisionNode
  dmCondEdge:ActEdge
  nextDmAct:Action
  dmEdge:ActEdge
  (actDM,decisionNode):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,decisionNode):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (actDM,nextDmAct):ContainsNode
  (actDM,dmCondEdge):ContainsEdge
  (dmCondEdge,nextDmAct):ConnectsTo
  (dmCondEdge,decisionNode):ConnectsFrom
  (dmCondEdge,guardSnapshotDM):GuardEdge
  (nextDmAct,preSnapshotDM):Pre_SnapshotPattern
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [dmCondEdge.guard<>oclUndefined(String)]
  [nextDmAct.name<>oclUndefined(String)]}
```

```
checkCorr(
  ((Action)ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)ucAct,nextDmAct) in nextExec:ExecControl
  (nextExec,currExec):NextExec}
end
--------------------------------------------------------------------
rule nextDMCondBranch_NewActGrp
checkSource(
  ucAct:SystemAction
){
}checkTarget(
  dmAct:Action
  actDM:Activity
  superActGrp:ActGroup
  objNode:ObjectNode
  preSnapshotDM:SnapshotPattern
  postSnapshotDM:SnapshotPattern
  guardSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
  [preSnapshotDM.isValid()]
  [guardSnapshotDM.isValid()]
){
  decisionNode:DecisionNode
  dmCondEdge:ActEdge
  nextDmAct:Action
  dmEdge:ActEdge
  actGrp:ActGroup
  (actDM,decisionNode):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,decisionNode):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (actDM,nextDmAct):ContainsNode
  (actDM,dmCondEdge):ContainsEdge
  (dmCondEdge,nextDmAct):ConnectsTo
  (dmCondEdge,decisionNode):ConnectsFrom
  (dmCondEdge,guardSnapshotDM):GuardEdge
  (actGrp,objNode):ActGroup_Object
  (actGrp,nextDmAct):GroupsNode
  (actGrp,dmCondEdge):GroupsEdge
  (superActGrp,actGrp):SubGroup
```

```
  (nextDmAct,preSnapshotDM):Pre_SnapshotPattern
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [dmCondEdge.guard<>oclUndefined(String)]
  [actGrp.opName<>oclUndefined(String)]
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  ((Action)ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)ucAct,nextDmAct) in nextExec:ExecControl
  (nextExec,currExec):NextExec}
end
-----------------------------------------------------------------------
rule nextDMCondBranch_ExtUC
checkSource(
  ucAct:SystemAction
  extendedUC:UseCase
  extensionUC:UseCase
  extCondSnapshot:SnapshotPattern
  [extCondSnapshot.isValid()]
){
  extPoint:ExtensionPoint
  extend:Extend
  (extend,extPoint):Extend_ExtensionPoint
  (extendedUC,extPoint):UC_ExtensionPoint
  (extensionUC,extend):UC_Extend
  (extend,extendedUC):Extend_UC
  (ucAct,extPoint):Action_ExtensionPoint
  (extPoint,extCondSnapshot):ExtCond}
checkTarget(
  dmAct:Action
  actDM:Activity
  superActGrp:ActGroup
  objNode:ObjectNode
  preSnapshotDM:SnapshotPattern
  postSnapshotDM:SnapshotPattern
  guardSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
  [preSnapshotDM.isValid()]
  [guardSnapshotDM.isValid()]
){
  decisionNode:DecisionNode
```

```
  dmCondEdge:ActEdge
  nextDmAct:Action
  dmEdge:ActEdge
  actGrp:ActGroup
  (actDM,decisionNode):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,decisionNode):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (actDM,nextDmAct):ContainsNode
  (actDM,dmCondEdge):ContainsEdge
  (dmCondEdge,nextDmAct):ConnectsTo
  (dmCondEdge,decisionNode):ConnectsFrom
  (dmCondEdge,guardSnapshotDM):GuardEdge
  (actGrp,objNode):ActGroup_Object
  (actGrp,nextDmAct):GroupsNode
  (actGrp,dmCondEdge):GroupsEdge
  (superActGrp,actGrp):SubGroup
  (nextDmAct,preSnapshotDM):Pre_SnapshotPattern
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [dmCondEdge.guard<>oclUndefined(String)]
  [actGrp.opName<>oclUndefined(String)]
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  ((Action)ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)ucAct,nextDmAct) in nextExec:ExecControl
  (extPoint,actGrp) as (extPoint,actGrp)
    in extPoint2ActGrp:ExtPoint2ActGrp
  (nextExec,currExec):NextExec}
end
-----------------------------------------------------------------------
rule nextUCAct
checkSource(
  ucAct:Action
  actUC:Activity
  preSnapshotUC:SnapshotPattern
  postSnapshotUC:SnapshotPattern
  (actUC,ucAct):ContainsNode
  [ucAct.postSnapshot.isValid()]
  [preSnapshotUC.isValid()]
){
  includedUcAct:UcAction
```

```
  ucEdge:ActEdge
  (actUC,includedUcAct):ContainsNode
  (actUC,ucEdge):ContainsEdge
  (ucEdge,includedUcAct):ConnectsTo
  (ucEdge,ucAct):ConnectsFrom
  (includedUcAct,preSnapshotUC):Pre_SnapshotPattern
  (includedUcAct,postSnapshotUC):Post_SnapshotPattern
  [includedUcAct.name<>oclUndefined(String)]}
checkTarget(
  dmAct:Action
  actDM:Activity
  objNode:ObjectNode
  preSnapshotDM:SnapshotPattern
  postSnapshotDM:SnapshotPattern
  (actDM,dmAct):ContainsNode
  [dmAct.postSnapshot.isValid()]
  [preSnapshotDM.isValid()]
){
  nextDmAct:Action
  dmEdge:ActEdge
  actGrp:ActGroup
  (actDM,nextDmAct):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,nextDmAct):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom
  (actGrp,objNode):ActGroup_Object
  (actGrp,nextDmAct):GroupsNode
  (actGrp,dmEdge):GroupsEdge
  (nextDmAct,preSnapshotDM):Pre_SnapshotPattern
  (nextDmAct,postSnapshotDM):Post_SnapshotPattern
  [actGrp.opName<>oclUndefined(String)]
  [nextDmAct.name<>oclUndefined(String)]}
checkCorr(
  (ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)includedUcAct,nextDmAct) in nextExec:ExecControl
  (includedUcAct,actGrp) as (ucAct,actGrp)
    in ucAct2ActGrp:UcAct2ActGrp
  (nextExec,currExec):NextExec}
end
-----------------------------------------------------------------------
rule finishScenario
```

```
checkSource(
  ucAct:Action
  actUC:Activity
  guardSnapshotUC:SnapshotPattern
  [guardSnapshotUC.isValid()]
){
  actFinishUC:FinalAction
  ucEdge:ActEdge
  (actUC,actFinishUC):ContainsNode
  (actUC,ucEdge):ContainsEdge
  (ucEdge,actFinishUC):ConnectsTo
  (ucEdge,ucAct):ConnectsFrom
  (ucEdge,guardSnapshotUC):GuardEdge}
checkTarget(
  dmAct:Action
  actDM:Activity
  [dmAct.postSnapshot.isValid()]
){
  actFinishDM:FinalAction
  dmEdge:ActEdge
  (actDM,actFinishDM):ContainsNode
  (actDM,dmEdge):ContainsEdge
  (dmEdge,actFinishDM):ConnectsTo
  (dmEdge,dmAct):ConnectsFrom  }
checkCorr(
  (ucAct,dmAct) as (ucAct,dmAct) in currExec:ExecControl
  [currExec.next=oclUndefined(ExecControl)]
){
  ((Action)actFinishUC,(Action)actFinishDM) in nextExec:ExecControl
  (nextExec,currExec):NextExec
  (nextExec,nextExec):NextExec}
end
```

Bibliography

[AJI04] Jesús Manuel Almendros-Jiménez and Luis Iribarne. Describing Use Cases with Activity Charts. In Uffe Kock Wiil, editor, *Metainformatics, International Symposium, MIS 2004, Salzburg, Austria, September 15-18, 2004, Revised Selected Papers*. Springer, LNCS 3511, 2004.

[AK03] C. Atkinson and T. Kuhne. Model-Driven Development: A Metamodeling Foundation. *Software, IEEE*, 20(5):36–41, 2003.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066, pages 361–375. Springer Berlin, 2006.

[ARNRSG06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model Traceability. *IBM Systems Journal*, 45, 2006.

[BBG+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *LNCS*. Springer, 2006.

[BG06] Fabian Büttner and Martin Gogolla. Realizing Graph Transformations by Pre- and Postconditions and Command Sequences. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, pages 398–413. Springer Berlin, LNCS 4178, 2006.

[BKPPT01] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A Visualization of OCL Using Collaborations. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *LNCS*, pages 257–271. Springer, 2001.

[BS06] Luciano Baresi and Paola Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006 Natal, Rio Grande do Norte, Brazil, September 17-23, 2006 Proceedings*, volume 4178, pages 306–320. Springer, 2006.

[Bé05] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling (SOSYM)*, 4(2):171–188, 2005.

[CCGdL08] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Analysing Graph Transformation Rules through OCL. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *LNCS*, pages 229–244. Springer, 2008.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. http://www.softmetaware.com/oopsla2003/mda-workshop.html, 2003.

[CJKW07] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, June 2007.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 1st edition, 2000.

[Dan07] Duc-Hanh Dang. Validation of System Behavior Utilizing an Integrated Semantics of Use Case and Design Models. In Claudia Pons, editor, *Proceedings of the Doctoral Symposium at the ACM/IEEE 10th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2007)*, volume 262, pages 1–5. CEUR, 2007.

[Dan08] Duc-Hanh Dang. Triple Graph Grammars and OCL for Validating System Behavior. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *LNCS*, pages 481–483. Springer, 2008.

[DBGP04] Amador Durán, Beatriz Bernárdez, Marcela Genero, and Mario Piattini. Empirically Driven Use Case Metamodel Evolution. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*, pages 1–11. Springer, LNCS 3273, 2004.

[DG09a] Duc-Hanh Dang and Martin Gogolla. On Integrating OCL and Triple Graph Grammars. In M.R.V Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, volume 5421, pages 124–137. Springer, 2009.

[DG09b] Duc-Hanh Dang and Martin Gogolla. Precise Model-Driven Transformation Based on Graphs and Metamodels. In *7th IEEE International Conference on Software Engineering and Formal Methods, 23-27 November, 2009, Hanoi, Vietnam*, pages 1–10. IEEE Computer Society Press, 2009. To appear.

[DG09c] Duc-Hanh Dang and Martin Gogolla. Towards Precise Operational Semantics of Modeling Languages on the Basis of TGGs and OCL. Technical Report AGDB-06-2009 (25 pages), University of Bremen, 2009.

[DGB07] Duc-Hanh Dang, Martin Gogolla, and Fabian Büttner. From UML Activity Diagrams to CSP Processes: Realizing Graph Transformations in the UML and OCL Tool USE. In *AGTIVE 2007 Tool Contest, 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance*, pages 1–25. http://gtcases.cs.utwente.nl/wiki/UMLToCSP/USE, 2007.

[dLBE+07] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Attributed Graph Transformation with Node Type Inheritance. *Theoretical Computer Science*, 376(3):139–163, May 2007.

[DLMP04] Isabel Diaz, Francisca Losavio, Alfredo Matteo, and Oscar Pastor. A Specification Pattern for Use Cases. *Information & Management*, 41(8):961–975, November 2004.

[dLV02] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer-Verlag, 2002.

[EEH08] Hartmut Ehrig, Claudia Ermel, and Frank Hermann. On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars. In *Proceedings of the third international workshop on Graph and model transformations*, pages 9–16. ACM, Leipzig, Germany, 2008.

[EFLR99] Andy Evans, Robert B. France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language. UML98: Beyond the Notation First International Workshop, Mulhouse, France, June 3-4, 1998. Selected Papers*, volume 1618 of *LNCS*, pages 336–348. Springer-Verlag, 1999.

[EH86] Hartmut Ehrig and Annegret Habel. Graph Grammars with Application Conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer-Verlag, 1986.

[EHK+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. *Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach.* World Scientific Publishing Co., Inc., 1997.

[EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 1, 2004. Proceedings*, volume 3256 of *LNCS*, pages 161–177. Springer Berlin, 2004.

[FR07] Robert France and Bernhard Rumpe. *Model-driven Development of Complex Software: A Research Roadmap.* IEEE Computer Society, 2007. ISBN 0-7695-2829-5.

[Fra03] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley, 2003.

[FV07] Marcos Didonet Del Fabro and Patrick Valduriez. Semi-Automatic Model Integration using Matching Transformations and Weaving Models. In Yookun Cho, Roger L. Wainwright, Hisham Haddad,

Sung Y. Shin, and Yong Wan Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pages 963–970. ACM, 2007.

[GBD08] Martin Gogolla, Fabian Büttner, and Duc-Hanh Dang. From Graph Transformation to OCL using USE. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited*, volume 5088 of *LNCS*, pages 585–586. Springer, 2008.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 2007.

[GdL06a] Esther Guerra and Juan de Lara. Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach. Technical Report UC3M-TR-CS-06-01 (61 pages), Universidad Carlos III de Madrid, 2006.

[GdL06b] Esther Guerra and Juan de Lara. Model View Management with Triple Graph Transformation Systems. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006 Natal, Rio Grande do Norte, Brazil, September 17-23, 2006 Proceedings*, volume 4178 of *LNCS*, pages 351–366. Springer, 2006.

[GGL05] Lars Grunske, Leif Geiger, and Michael Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In Alan Hartman and David Kreische, editors, *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings*, volume 3748 of *LNCS*, pages 284–298. Springer, 2005.

[GK07] Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *LNCS*, pages 16–30. Springer, 2007.

[GKB08] Martin Gogolla, Mirco Kuhlmann, and Fabian Büttner. A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and

Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *LNCS*, pages 446–459. Springer, 2008.

[GLST01] Wolfgang Grieskamp, Markus Lepper, Wolfram Schulte, and Nikolai Tillmann. Testable Use Cases in the Abstract State Machine Language. In *2nd Asia-Pacific Conference on Quality Software (APAQS 2001), 10-11 December 2001, Hong Kong, China, Proceedings*, pages 167–172. IEEE Computer Society, 167-172, 2001.

[GNE+08] Javier Gutiérrez, Clémentine Nebut, María Escalona, Manuel Mejías, and Isabel Ramos. Visualization of Use Cases through Automatically Generated Activity Diagrams. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *LNCS*, pages 83–96. Springer, 2008.

[GR97] Martin Gogolla and Mark Richters. On Constraints and Queries in UML. In Martin Schader and Axel Korthaus, editors, *Proc. UML'97 Workshop 'The Unified Modeling Language - Technical Aspects and Applications'*, pages 109–121. Physica-Verlag, Heidelberg, 1997.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 1st edition, August 2004.

[GW06] Holger Giese and Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *LNCS*, pages 543–557. Springer, 2006.

[HH01] Jan Hendrik Hausmann and Reiko Heckel. Use Cases as Views: A Formal Approach to Requirements Engineering in the Unified Process. In *GI Jahrestagung*, pages 595–599, 2001.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[HHT02] Jan Hendrik Hausmann, Reiko Heckel, and Gabriele Taentzer. Detection of Conflicting Functional Requirements in a Use Case-Driven

Approach: A Static Analysis Technique Based on Graph Transformation. In *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. ACM, 2002.

[HKT02a] Reiko Heckel, Jochen Küster, and Gabriele Taentzer. Towards Automatic Translation of UML Models into Semantic Domains. In H.-J. Kreowski and P. Knirsch, editors, *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002), Grenoble, France, 2002*, pages 11–22. http://www.informatik.uni-bremen.de/theorie/AGT2002/, 2002.

[HKT02b] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *LNCS*, pages 161–176. Springer Berlin, 2002.

[Hur97] Russell R. Hurlbut. A Survey of Approaches for Describing and Formalizing Use Cases. Technical Report XPT-TR-97-03, Department of Computer Science, Illinois Institute of Technology, USA, 1997.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, USA, 1st edition, June 1992. ISBN 0201544350.

[JLMT08] Stefan Jurack, Leen Lambers, Katharina Mehner, and Gabriele Taentzer. Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *LNCS*, pages 341–355. Springer, 2008.

[KM08] Pierre Kelsen and Qin Ma. A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In *Model Driven Engineering Languages and Systems, 11th International Conference,*

MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings, volume 5301, pages 690–704. Springer Berlin, 2008.

[KR06] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, Lecture Notes in Computer Science, pages 299–305. Springer, 2006.

[Kre93] Hans-Jörg Kreowski. Five Facets of Hyperedge Replacement Beyond Context-Freeness. In Zoltán Ésik, editor, *Proceedings of the 9th International Symposium on Fundamentals of Computation Theory*, volume 710 of *LNCS*, pages 69–86. Springer-Verlag, 1993.

[KRW04] E. Kindler, V. Rubin, and R. Wagner. An Adaptable TGG Interpreter for In Memory Model Transformations. In *2nd International Fujaba Days 2004 at Darmstadt Technical University, Germany. Proceedings*, pages 35–38. http://www.fujaba.de/fdays, 2004.

[KS04] Alexander Königs and Andy Schürr. Multi-Domain Integration with MOF and extended Triple Graph Grammars. In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development, 29. February - 5. March 2004*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[KS06] Alexander Königs and Andy Schürr. Tool Integration with Triple Graph Grammars - A Survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113–150, February 2006.

[KSLB03] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[Kur08] Ivan Kurtev. State of the Art of QVT: A Model Transformation LanguageStandard. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited*, volume 5088 of *LNCS*, pages 585–586. Springer, 2008.

[KW07] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations and Application Scenarios. Technical Report tr-ri-07-284, University of Paderborn, Germany, 2007.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture(TM): Practice and Promise*. Addison-Wesley Professional, May 2003.

[Kö05] Alexander Königs. Model Transformation with Triple Graph Grammars. In L.C. Briand and C. Williams, editors, *Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica, 2005*, volume 4199 of *LNCS*. Springer, 2005.

[Kü06] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[LLC08] Laszlo Lengyel, Tihamer Levendovszky, and Hassan Charaf. Validated model transformation-driven software development. *Int. J. Comput. Appl. Technol.*, 31(1/2):106–119, 2008.

[LLVC06] Laszlo Lengyel, Tihamér Levendovszky, Tamás Vajk, and Hassan Charaf. Realizing QVT with Graph Rewriting-Based Model Transformation. *Electronic Communications of the EASST*, 4(Graph and Model Transformation 2006), 2006.

[LS06] Alexander Lorenz and Hans-Werner Six. Tailoring UML Activities to Use Case Modeling for Web Application Development. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 26. ACM Press, Toronto, Ontario, Canada, 2006.

[Lö93] Michael Löwe. Algebraic Approach to Single-Pushout Graph Transformation. *Theor. Comput. Sci.*, 109(1-2):181–224, 1993.

[MBMB02] Stephen J. Mellor, Marc J. Balcer, Stephen Mellor, and Marc Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley Professional, 1st edition, May 2002. ISBN 0201748045.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems*, volume 3713, pages 264–278. Springer Berlin, 2005.

[MSD+08] Tanveer Mustafa, Karsten Sohr, Duc-Hanh Dang, Michael Drouineaud, and Stefan Kowski. Implementing Advanced RBAC Administration Functionality with USE. In *Proc. 8th Workshop on OCL at the ACM/IEEE 11th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2008)*, volume 15, pages 1–19. Electronic Communications of the EASST, 2008.

[NFTJ06] C Nebut, F Fleurey, Y Le Traon, and J Jezequel. Automatic Test Generation: A Use Case Driven Approach. *Software Engineering, IEEE Transactions on*, 32(3):140–155, 2006.

[OMG03a] OMG. *Common Warehouse Metamodel (CWM) Specification*. OMG, 2003.

[OMG03b] OMG. *MDA Guide Version 1.0.1*. OMG, 2003.

[OMG06] OMG. *Object Constraint Language OMG Available Specification Version 2.0*. OMG, 2006.

[OMG07a] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification ptc/07-07-07*. OMG, 2007.

[OMG07b] OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*. OMG, November 2007.

[OMG07c] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. OMG, November 2007.

[PM03] Frantisek Plasil and Vladimir Mencl. Getting 'Whole Picture' Behavior in a Use Case Model. In *Proceedings of IDPT 2003*. Society for Design and Process Science, Grandview, Texas, Austin, TX, 2003.

[Pra71] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Academic Press*, 5(6):560–595, 1971.

[RAB96] Björn Regnell, Michael Andersson, and Johan Bergstrand. A Hierarchical Use Case Model with Graphical Representation. In *IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96), March 11-15, 1996, Friedrichshafen, Germany*, page 270. IEEE Computer Society, 1996.

[RB03] Kexing Rui and Greg Butler. Refactoring Use Case Models: the Metamodel. In *Proceedings of the 26th Australasian computer science conference*, pages 301–308. Australian Computer Society, Inc., Adelaide, Australia, 2003.

[RG98] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong-Li Lee, editors, *Conceptual Modeling - ER '98, 17th International Conference on Conceptual Modeling, Singapore, November 16-19, 1998, Proceedings*, volume 1507 of *Lecture Notes in Computer Science*, pages 449–464. Springer, 1998.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, 2002.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual, 2nd Edition*. Addison-Wesley Professional, 2004.

[Rot89] Jeff Rothenberg. The Nature of Modeling. In Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen, editors, *Artificial Intelligence, Simulation & Modeling*, pages 75–92. New York, John Wiley and Sons, Inc., 1989.

[SBN+07] Michal Smialek, Jacek Bojarski, Wiktor Nowakowski, Albert Ambroziewicz, and Tomasz Straszak. Complementary Use Case Scenario Representations Based on Domain Vocabularies. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, pages 544–558. Springer Berlin, LNCS 4735, 2007.

[Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Mayr Schmidt, editor, *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. Springer-Verlag, 1995.

[Sei03] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[Sen02] Shane Sendall. *Specifying Reactive System Behavior*. PhD thesis, Swiss Federal Institute of Technology in Lausanne, School of Computer and Communication Sciences, 2002.

[SK03] Shane Sendall and W. Kozaczynski. Model Transformation: the Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42– 45, 2003. ISSN 0740-7459.

[SPW07] Avik Sinha, Amit Paradkar, and Clay Williams. On Generating EFSM Models from Use Cases. In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, page 97. IEEE Computer Society, 2007.

[SWZ96] Andy Schürr, Andreas Winter, and Albert Zündorf. Developing Tools with the PROGRES Environment. In Manfred Nagl, editor, *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *LNCS*, pages 356–369. Springer, 1996.

[SZG06] Mirko Stölzel, Steffen Zschaler, and Leif Geiger. Integrating OCL and Model Transformations in Fujaba. In Dan Chiorean, Birgit Demuth, Martin Gogolla, and Jos Warmer, editors, *Proceedings of the Sixth OCL Workshop OCL for (Meta-) Models in Multiple Application Domains (OCLApps2006)*, volume 5 of *OCL Workshop*. Electronic Communications of the EASST, 2006.

[VAB+08] Dániel Varró, Márk Asztalos, Dénes Bisztray, Artur Boronat, Duc-Hanh Dang, Rubino Geiß, Joel Greenyer, Pieter Van Gorp, Ole Kniemeyer, Anantha Narayanan, Edgars Rencis, and Erhard Weinell. Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited*, volume 5088 of *LNCS*, pages 540–565. Springer, 2008.

[VFV06] Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a Graph Transformation Engine in Relational Databases. *Software and Systems Modeling*, 5(3):313–341, 2006.

[VP03] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics). *Software and Systems Modeling*, 2(3):187–210, October 2003.

[Wag06] Robert Wagner. Developing Model Transformations with Fujaba. In Holger Giese and Bernhard Westfechtel, editors, *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275. University of Paderborn, 2006.

[Whi06] Jon Whittle. Specifying Precise Use Cases with Use Case Charts. In Jean-Michel Bruel, editor, *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*, pages 290–301. Springer, LNCS 3844, 2006.

[WK98] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Addison-Wesley Professional, 1st edition, 1998.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA (2nd Edition)*. Addison-Wesley Professional, 2 edition, 2003.

[ZG03] Paul Ziemann and Martin Gogolla. OCL Extended with Temporal Logic. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International AndreiErshov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk,Russia, July 9-12, 2003, Revised Papers*, volume 2890, pages 617–633. LNCS, 2003.

[ZHG05] Paul Ziemann, Karsten Hölscher, and Martin Gogolla. From UML Models to Graph Transformation Systems. *Electr. Notes Theor. Comput. Sci.*, 127(4):17–33, 2005.

[Zie05] Paul Ziemann. *An Integrated Operational Semantics for a UML Core Based on Graph Transformation*. PhD thesis, Department of Computer Science, University of Bremen, Germany, 2005.

List of Figures