FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Testing and Debugging of Model Transformations

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

DI (FH) Johannes Schönböck
Matrikelnummer 0057399

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel

Diese Dissertation haben begutachtet:

(o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel)	(Associate Professor Ph.D. Juan De Lara)

Wien, 22.12.2011

(DI (FH) Johannes Schönböck)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Testing and Debugging of Model Transformations

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

DI (FH) Johannes Schönböck
Registration Number 0057399

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel

The dissertation has been reviewed by:

(o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel)	(Associate Professor Ph.D. Juan De Lara)

Wien, 22.12.2011

(DI (FH) Johannes Schönböck)

Erklärung zur Verfassung der Arbeit

DI (FH) Johannes Schönböck
Lina 1, 4311 Schwertberg, Austria

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

The completion of this thesis represents surely a major point in my academic career. Nevertheless, although the actual writing of a thesis is individual work, the research and foundations underlying this thesis could only be achieved by an excellent teamwork. Therefore, in a first step I deeply want the thank my colleague Angelika Kusel. Working with her was always a pleasure and her cooperative and conscientious manner lead to memorable research results. Thank you very much for motivating me during these three years. Of course, this would not have been possible without our always cooperative and helpful supervisors Gerti Kappel, Werner Retschitzegger, Wieland Schwinger and Manuel Wimmer. Without the numerous common discussions, as well as their ideas and comments, I would not have been able to deliver this kind of work. Whenever having any kind of troubles they supported me in finding a solution. Thank you for the good atmosphere and working environment making the hard work of doing the thesis a pleasure.

Additionally, I want to thank my reviewer Juan de Lara. When I first met him on a conference, I was not only impressed by his excellent work in the area of model transformations, but also by his cooperative manner. Thus, I have been very pleased that he agreed to act as an external reviewer of my thesis. His valuable comments helped me a lot to incorporate new ideas and consequently helped me to improve the outcome of the thesis. In this respect, I would like also to thank Esther Guerra for having the pleasure to work with her together on the topic of model transformation contracts and QVT Relations.

I am very much obliged to our secretaries Birgit Hauer and Katja Hildebrandt, who were always supportive in the non-academic issues. Furthermore, I appreciate very much that Katharina Kapplmüller helped me to improve the English by proof-reading the thesis. I also want to express my gratitude to my colleagues Petra Brosch, Philip Langer and Konrad Wieland, who often served as valuable discussion partners and with whom I spent many placid hours on conferences.

By writing these words I can by no means express how much I need to thank my family, my mother Rosa, my dad Johann, and my brother Reinhard. They always stood by my side and supported me in finishing my thesis. They enabled my education, trusted in me and supported me during my whole life – thank you very much. Last but not least, I want to thank all of my friends who encouraged me in finishing my thesis.

Abstract

Model-Driven Engineering (MDE) proposes an active use of models to conduct the different phases of software development. The major vision is a shift from the idea of "everything is an object" in the object-oriented paradigm to the idea of "everything is a model" in MDE. Following this vision, it becomes obvious that transformations between models play a key role. Just like any other software, transformations should be engineered using sound and robust engineering techniques. However, current engineering techniques focus on the implementation phase of transformations, but fail to provide means for the analysis, design, testing and debugging phases.

In particular, to support the analysis and design phase, means are needed that allow to formally describe the requirements of a certain transformation in order to allow for automatic validation in the testing phase. In case of a failure, additional means are needed to efficiently debug model transformations. However, current transformation languages provide only scarce support for debugging. This is mainly due to the fact that low-level information of an according execution engine is provided only, e.g., variable values. Finally, the operational semantics is hidden by these execution engines, which further aggravates finding failures and hampers understanding of transformation specifications.

To tackle the aforementioned limitations, this thesis provides three main contributions. First, a declarative, visual language called PAMOMO is proposed, which allows to formally specify requirements on model transformations by means of contracts. To test if a model transformation fulfills the specified requirements, the contracts are compiled into check-only QVT Relations, providing dedicated error traces in case a contract fails. These traces may then be used as hints for debugging. To support debugging, Transformation Nets as a DSL on top of CPNs are proposed, which provide a dedicated runtime model for model transformations, making the hidden operational semantics explicit as a second major contribution. Finally, based on this runtime model various means of debugging are presented as a third contribution.

To evaluate the contributions, relations to competing approaches are drawn in a first step. Second, case studies are used to show the applicability of the presented approaches. To evaluate the runtime model, the operational semantics of dedicated transformation languages is made explicit in terms of Transformation Nets. Finally, the debugging support is evaluated again by case studies and a first user study.

Kurzfassung

Modellgetriebene Softwareentwicklung rückt Modelle ins Zentrum des Softwareentwicklungsprozesses. Dadurch nehmen Modelle die Rolle von Objekten in der objektorientierten Softwareentwicklung ein. Durch diese zentrale Rolle entsteht die Notwendigkeit Transformationen zwischen Modellen durchzuführen. Analog zur traditionellen Softwareentwicklung sollen Modelltransformationen auf fundierte Sprachen und Werkzeuge zurück greifen können. Aktuelle Transformationssprachen fokussieren allerdings nur auf die Implementierungsphase und berücksichtigen weitere Phasen wie Analyse, Design, Testen und Fehlersuche nur unzureichend.

Für die Analyse- und Designphase werden Mittel benötigt, die es dem Transformationsentwickler erlauben, die Anforderungen formal zu spezifizieren und diese dann in der Testphase gegen die implementierte Transformation zu validieren. Für den Fall, dass Anforderungen nicht erfüllt sind, werden Werkzeuge und Mechanismen zur Fehlersuche benötigt. Aktuell verwendete Transformationssprachen bieten hierbei aber nur unzureichende Unterstützung, da sie nur Informationen bereit stellen, die von den jeweiligen Laufzeitumgebungen zur Verfügung gestellt werden. Da diese typischerweise in einer Programmiersprache wie z.B. Java entwickelt sind, bestehen solche Informationen meist nur aus Werten von Variablenbelegungen. Des Weiteren verstecken die auf niedrigem Abstraktionsniveau arbeitenden Laufzeitumgebungen die Ausführungssemantik der Transformation, was die Fehlersuche zusätzlich erschwert.

Um diese Einschränkungen aufzuheben, werden im Rahmen der Arbeit drei Hauptbeiträge erarbeitet. Als erster Beitrag wird die deklarative Sprache PAMOMO vorgestellt, die eine Spezifikation von Transformations-Kontrakten ermöglicht. Um zu testen, ob Transformationen die Kontrakte erfüllen, wird QVT Relations verwendet, um im Fehlerfall Information zu erhalten, die im weiterem für die Fehlersuche verwendet werden kann. Transformationsnetze stellen als zweiten Hauptbeitrag ein explizites Laufzeitmodell für Transformationen zur Verfügung und legen dadurch deren operationale Semantik offen. Dieses Laufzeitmodell bildet damit die Grundlage für Methoden zur Unterstützung bei der Fehlersuche.

Zur Evaluierung der Arbeit werden Vergleiche zu bestehenden Arbeiten gezogen. Mittels Fallbeispielen wird die Anwendbarkeit der vorgestellten Konzepte gezeigt. Um die Laufzeitumgebung zu evaluieren, wird die Ausführungssemantik existierender Transformationssprachen auf Transformationsnetze abgebildet. Dadurch können auch diese Sprachen von den vorgeschlagenen Methoden zur Fehlersuche profitieren, was wiederum mittels Fallbeispielen gezeigt wird.

Preface

The research presented in this thesis was undertaken at the Institute of Software Technology and Interactive Systems, Business Informatics Group (BIG), Vienna University of Technology, in joint work with DI Angelika Kusel from the Institute of Bioinformatics, Working Group Information Systems (IFS), Johannes Kepler University, Linz, and was partially funded by the Austrian Science Fund under grant P21374-N13 (cf. project TROPIC[1]). The supervisors comprise Prof. Dr. Gerti Kappel and Dr. Manuel Wimmer from Vienna and Prof. Dr. Werner Retschitzegger and Prof. Dr. Wieland Schwinger from Linz. The major concepts and techniques developed during my thesis have been peer-reviewed and published in international conference proceedings and international workshop proceedings, resulting in a list of publications as detailed below:

1. "Lets's Play the Token Game – Model Transformations Powered By Transformation Nets", Co-Autors: M. Wimmer, A. Kusel, T. Reiter, W. Retschitzegger, and W. Schwinger, in Proceedings of the International Workshop on Petri Nets and Software Engineering, (PNSE), in conjunction with 30th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency, Paris, France, June 22-23, pp. 35-50, 2009.

2. "Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets", Co-Autors: M. Wimmer, A. Kusel, G. Kappel, W. Retschitzegger, and W. Schwinger, in Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09), Denver, Colorado, USA, October 4-9, Springer, pp. 727-732, 2009.

3. "Transformation Nets - A Runtime Model for Transformation Languages", in Proceedings of Doctoral Symposium at ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems, Denver, Colorado, USA, October 4-9, pp. 28-34, 2009

4. "Catch me if you can - Debugging Support for Model Transformations", Co-Autors: A. Kusel, G. Kappel, W. Retschitzegger, W. Schwinger, and M. Wimmer, in Proceedings of

[1] http://www.modeltransformation.net

Models in Software Engineering, Workshops and Symposia at MoDELS 2009, Reports and Revised Selected Papers, Springer-Verlag, pp. 5-20, 2010.

5. "Right or Wrong? – Verification of Model Transformations using Colored Petri Nets", Co-Autors: M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, in Proceedings of the 9th Workshop on Domain-Specific Modeling (DSM), in conjunction with 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09), Orlando, Florida, USA, October 25-29, Helsinki Business School, 2009.

6. "TROPIC: A Framework for Model Transformations on Petri Nets in Color", Co-Authors: M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, and W. Schwinger, in Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09), Orlando, Florida, USA, October 25-29, ACM, pp. 783-784, 2009.

7. "A Petri Net based Debugging Environment for QVT Relations", Co-Authors: M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, and W. Schwinger, in Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09), Aukland, New Zealand; November 16-20, IEEE, pp. 1-12, 2009.

8. "Taming the Shrew – Resolving Structural Heterogeneities with Hierarchical CPNs", Co-Authors: M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, and W. Schwinger, in Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE), in conjunction with 31th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency, Braga, Portugal, June 21-25, University of Hamburg, pp. 141-157, 2010.

9. "Surviving the Heterogeneity Jungle with Composite Mapping Operators", Co-Authors: M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, and W. Schwinger, in Proceedings of the 3rd International Conference on Model Transformation (ICMT'10), Malaga, Spain, June 28-July 2, Springer-Verlag, pp. 260-275, 2010.

10. "On using Inplace Transformations for Model Co-evolution", Co-Authors: M. Wimmer, A. Kusel, W. Retschitzegger, W. Schwinger, and G. Kappel, in Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL), in conjunction with 3rd International Conference on Model Transformation (ICMT'10), Malaga, Spain, June 28-July 2, INRIA & École des Mines de Nantes, 2010.

11. "Plug & Play Model Transformations – A DSL for Resolving Structural Metamodel Heterogeneities", Co-Authors: M. Wimmer, G.Kappel, W. Retschitzegger, J. Schönböck, and W. Schwinger, in Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM'10), in conjunction with Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'10), Reno/Tahoe Nevada, USA, October 17-21, Online Publication, 2010.

12. “Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities”, Co-Authors: M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, and W. Schwinger, in Proceedings of the First International Workshop on Model-Driven Interoperability (MDI), in conjunction with 13th International Conference on Model Driven Engineering Languages and Systems(MoDELS’10), Oslo, Norway, October 3-8, ACM, pp. 32-41, 2010.

13. “From the Heterogeneity Jungle to Systematic Benchmarking”, Co-Authors: M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, and W. Schwinger, in Proceedings of Models in Software Engineering - Workshops and Symposia at MoDELS 2010, Reports and Revised Selected Papers, Springer-Verlag, pp. 150-164, 2010.

14. “A Comparison of Rule Inheritance in Model-to-Model Transformation Languages”, Co-Authors: M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, D. Kolovos, R. Paige, M. Lauder, A. Schürr, and D. Waagelar, in Proceedings of the 4th International Conference on Model Transformation (ICMT’11), Zurich, Switzerland, June 27-28, Springer-Verlag, pp. 31-46, 2011.

15. “Reusing Model Transformations across Heterogeneous Metamodels”, Co-Authors: M. Wimmer, A. Kusel, W. Retschitzegger, W. Schwinger, J. S. Cuadrado, E. Guerra, and J. de Lara, in Proceedings of the 5th International Workshop on Multi-Paradigm Modeling (MPM’10), in conjunction with 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS’11), Wellington, New Zealand, October 16-20, Springer-Verlag, 2011.

16. “Automated Verification of Model Transformations based on Visual Contracts”, Co-Authors: E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, accepted for publication in Journal for Automated Software Engineering.

Contents

List of Figures

Do or do not...
there is no try.

— *Yoda (Fictional character from George Lucas's Star Wars)*

Chapter 1

Introduction

It is not because things are difficult that we do not dare;
it is because we do not dare that they are difficult.

— *Lucius Annaeus Seneca*

Contents

1.1 Motivation

Abstraction has always been key in software engineering to deal with the omnipresent problem of growing *complexity*. In a first step, software engineers have tried to abstract from the underlying computing environment, e.g., CPU and memory, as stated in [133]. The development of high-level programming languages represented a major step in this direction. Nevertheless, abstraction mechanisms in dedicated programming languages did not raise the level of abstraction in the *design phase* of software but only in the *implementation phase* since their focus was on the *solution domain*, i.e., the programming languages, only. Therefore, technologies have been developed for raising the level of abstraction in the design phase already. A first prominent representative in this direction was *Computer-Aided Software Engineering* (CASE) [33]. Simply speaking, the main goal of CASE was to automatically generate executable code from graphical representations of a system, i.e., graphical programming. Although CASE attained attraction in

the research community, it has never been fully adopted in practice. This was the case because only very general-purpose graphical representations were given, e.g., state machines, structure diagrams, and dataflow diagrams which poorly mapped to the underlying platforms [133]. Moreover, due to a lack of commonly accepted middleware platforms all the necessary infrastructure code had to be generated as well, which made it difficult to integrate the generated code with other software. Consequently, CASE tools were mostly used to visualize the software architecture, acting as a guide for the actual manual implementation of the software only. Although nowadays more powerful frameworks and middleware platforms are available which might overcome some deficiencies of CASE tools, e.g., J2EE[1], .NET[2], CORBA[3], Eclipse Platform[4], and Spring[5], to mention just a few, software engineers are again confronted with growing complexity. Currently, researchers try to address this problem by providing *Domain-Specific Modeling Languages* (DSMLs) which are specifically tailored languages that fit into a certain problem domain, e.g., by employing *models* that are less bound to an underlying implementation technology and are much closer to the *problem domain*. Thus, not only software engineers may implement a system but domain experts are enabled to *model* a system by means of a DSML, which is key to the idea of *Model-Driven Engineering*, as explained in detail in the following.

1.1.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [133] proposes an active use of models to conduct the different phases of software development. Thus, models become first-class artifacts throughout the different phases of the software development life cycle. This leads to a shift from the "everything is an object" paradigm to the "everything is a model" paradigm [16]. Although models have been used in software engineering before MDE arose, models rather served for documentation purposes than as a program, i.e., models were not coupled with the according code. In the context of MDE, models are abstractions of systems [38] and serve as single source of information to specify the implementation of a system. Consequently, developers may focus on modeling a system close to the actual problem domain [138] and do not have to deal with the low-level details of an underlying implementation platform or a certain programming language. In the same way as programs have to follow certain syntactic constraints commonly described by a grammar, models also have to follow syntactic constraints given by metamodels which define the abstract syntax. Consequently, metamodels define concepts, their relationships as well as constraints among each other which are prevalent in a certain domain.

The Object Management Group[6] (OMG) standardized the concepts of MDE in their Model Driven Architecture initiative[7] [80, 104]. MDA especially focusses on providing so-called Platform Independent Models (PIMs), which allow the modeler to focus on the problem domain, and Platform Specific Models (PSMs), which enrich PIMs with platform-specific information.

[1] http://download.oracle.com/javaee
[2] http://www.microsoft.com/net/default.aspx
[3] http://www.corba.org
[4] www.eclipse.org
[5] http://www.springsource.org
[6] http://www.omg.org
[7] http://www.omg.org/mda/specs.htm

Therefore, PSMs are closer to a certain implementation platform. Besides models on different level of abstractions, *model transformations* between models are key to the success of MDE, e.g., to automatically transform PIMs to PSMs. Thus, in the following model transformations are discussed in more detail.

1.1.2 Model Transformations

Transformations between different artifacts are ubiquitous in software engineering, for example, in case of compiling high-level programs into low-level byte code and thus transformations in general are known from other engineering domains. Thus, model transformations are closely related to, e.g., data exchange in data engineering, when treating models as data and metamodels as schemas. A similar analogy might be drawn for the area of ontology engineering.

In the area of *data engineering*, the history of engineering transformations goes back to 1976. The inventor of the Entity-Relationship (ER) model, Peter Chen discussed the problem of generating suitable relational schemas out of ER models [34]. Furthermore, IBM suggested the EXPRESS (Extraction, Processing and Restructuring System) approach to transform data between hierarchical databases [142]. These two important publications were followed by a huge amount of work in different areas and application domains of information integration (cf., e.g., [60] or [62] for an overview). This work ranges from the area of multi- and federated database systems in the 1980s (cf., e.g., Sheth et al. [141]), data warehouses as well as the integration of non-relational sources in the 1990s [117], to efforts in the more recent past in the areas of schema matching (cf., e.g., [122]), generic model management (cf., e.g, [5], [14]), mapping tools (cf., e.g., [94], [121]) and most recently, data mashups and dataspaces [71].

The field of *ontology engineering* has to cope with similar transformation problems as common in the area of data engineering. For example different architectures have been proposed for the purpose of ontology integration. The approaches may be distinguished between direct mappings, indirect mappings via a common, shared ontology and mappings based on a library of already mapped ontologies [114]. Additionally, numerous approaches for the discovery, representation and reasoning of mappings have already emerged [75].

In the context of MDE, there is a need to transform models between different languages and abstraction levels, e.g., to migrate between language versions, to translate models into semantic domains for analysis, to generate PSMs from PIMs, and to refine and abstract models [47]. Thus, model transformations are comparable in role and importance to compilers for high-level programming languages, since models have to be automatically refined until the code of the final application is obtained. In order to describe how models should be transformed into other models, the transformation definition takes place between the respective *metamodels* the models conform to (cf. Fig. 1.1). To specify model transformations, dedicated transformation languages, especially tailored to the task of transforming models, exist (cf. [38] for an overview).

Thereby, transformation languages may be divided into *imperative*, *declarative* and *hybrid* approaches. Imperative approaches like Kermeta [110] and the OMG standard QVT Operational [116] allow for an easier specification of complex transformations than declarative approaches, e.g., by providing explicit statefulness, but inducing more overhead code, as many tasks have to be accomplished explicitly, e.g., the specification of the control flow. Declarative and hybrid transformation languages relieve transformation designers from this burden since it

is only necessary to specify *what* has to be transformed but not *how* this is done. The specification is done by means of declarative rules which are then executed by dedicated transformation engines (cf. Fig. 1.1). The OMG standard QVT Relational [116] represents a declarative approach whereas prominent hybrid representatives are, e.g., Atlas Transformation Language (ATL) [73], and Epsilon Transformation Language (ETL) [81]. Furthermore, *graph based* approaches, e.g., Triple Graph Grammar (TGG) [136], AGG [147], Atom3 [89], and Viatra [9], have been proposed, being either purely declarative or they additionally include means to specify control structures. These approaches are based on the fact that models may be represented as typed, attributed graphs [4]. These graphs may be modified using graph transformation rules which consist of a left-hand side (LHS) and a right-hand side (RHS) pattern. During execution, the LHS pattern is matched for the source model, and – if it is found – replaced by the RHS pattern (in its simplest form).

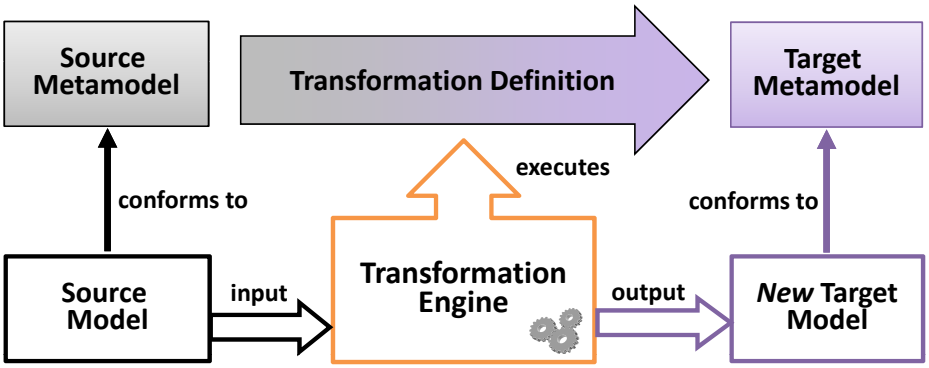


Figure 1.1: Basic Model-to-Model Transformation Pattern

Fig. 1.1 shows the basic model transformation pattern, where a source model conforming to a source metamodel is transformed into a new target model conforming to a target metamodel. This scenario is called a *batch and exogenous model-to-model transformation*. However, many other scenarios are possible as well (cf. [38]). First, the source model may change after the transformation is executed. In this case, it is sometimes more efficient not to build the target model from scratch but to *update* it. Then, transformations may also be *bidirectional*, if the same specification may be used to transform from source to target and the other way round. Transformations may also be used in *check-only* mode, for example, to ascertain whether two existing models comply with the transformation definition. Finally, a model may be transformed "in-place", for example, for refactoring. In this case, the transformation definition only considers one metamodel, and is called *endogenous*.

This thesis focuses on declarative, rule-based model-to-model transformation languages considering a batch and exogenous scenario. The presented concepts may nevertheless be adapted to other types of transformation languages and also to other scenarios.

1.2 Running Example

Before delving into details, a small extract of the `Class2Relational` transformation problem, which has been chosen as a running example due to its popularity in the scientific community, is introduced. This problem is used throughout this thesis (cf. Fig. 1.2) [18]. In this batch and exogenous model-to-model transformation, the goal is to transform instances of the `class` metamodel into instances of the `relational` metamodel, i.e., this example represents a unidirectional, batch and exogenous model-to-model transformation. In this context, six main requirements arise:

- *Requirement 1*: For each instance of the class `Package` a corresponding instance of the class `Schema` should be generated, which should be equally named (cf. instances `p1` and `s1` in Fig. 1.2).
- *Requirement 2*: For each instance of the class `Class`, which is persistent, a corresponding instance of the class `Table` should be generated, which should be equally named (cf. instances `c2`, `c3`, `t1`, and `t2` in Fig. 1.2).
- *Requirement 3*: For each instance of the class `Attribute`, belonging to a persistent `Class`, a corresponding instance of the class `Column` should be generated, which should be equally named (cf. instances `a2`, `a3`, `co1`, and `co3` in Fig. 1.2).

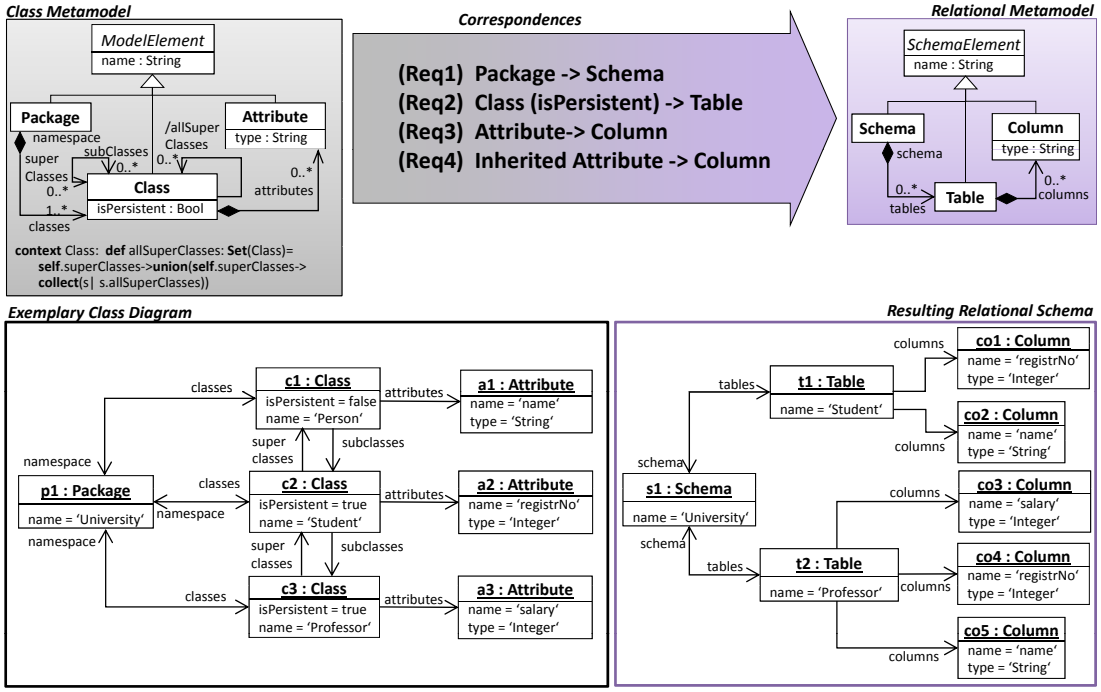


Figure 1.2: Running Example: Translating Class Diagrams into Relational Schemas

- *Requirement 4*: Since the relational metamodel does not support inheritance between instances of `Table` and since information loss should be prevented during the transformation process, for each inherited `Attribute` instance a corresponding `Column` instance should be generated (cf. instances `co2`, `co4`, and `co5` in Fig. 1.2).

Besides requirements that need to be satisfied by any pair of input/output models, requirements may exist that solely concern the input models. Such requirements are used to put *further constraints on input models* to exclude those not handled by the transformation although they conform to the source metamodel. This is due to the fact that metamodels allow in general for many different valid models, but a certain transformation definition might only cover a subset thereof. In the context of the example, a requirement of the input model is the following:

- *Requirement 5*: Class models are not allowed to contain redefined attributes, i.e., attributes with the same name in an inheritance hierarchy, since otherwise tables containing equally named columns would result.

Finally, a certain transformation might need to guarantee that the produced output models fulfill certain conditions beyond metamodel constraints. In the example, the following fact is demanded:

- *Requirement 6*: Relational models may not contain tables with equally named columns, even though this is allowed by the metamodel.

By current transformation engines only syntactical correctness is checked but it is left unclear if the posed requirements are accordingly regarded by the specification. In case the requirements are not fulfilled, additionally the question arises how to find an error in a transformation specification, i.e., which means for debugging are provided. In the following, deficiencies of model transformation languages concerning the specification of requirements as well as shortcomings of existing support for testing and debugging are shortly elaborated.

1.3 Deficiencies

Transformation languages focus on the implementation of transformations but fail to provide means for analysis, design, testing and debugging of model-to-model transformations [58, 98]. However, just like any other software, transformations should be engineered using sound, robust engineering techniques. This necessity is even more acute given the prominent role of transformations in MDE and their increasing complexity. Hence, the MDE community demands for methods and techniques supporting appropriate abstractions to be used in the different phases of transformation development (cf. Fig. 1.3). In the following, the current state of the art in the main phases as well as current deficiencies are described.

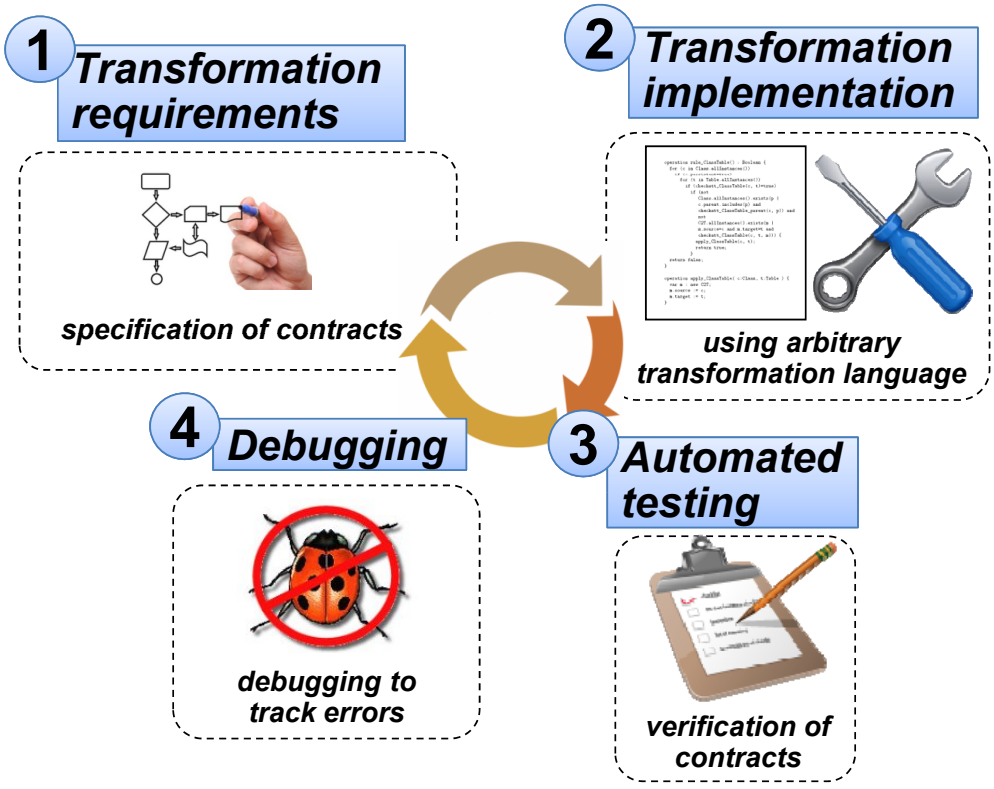


Figure 1.3: Model Transformation Development Phases

1.3.1 Requirements and Analysis Phase

In software engineering, the determination of requirements and their analysis has been recognized as an important part of the whole engineering process, reflected by the dedicated discipline of *requirements engineering* [130]. In research as well as in industry it is widely accepted that an accurate requirement analysis is critical to the success of a software project. Concerning model transformations, however, the sound specification of requirements is still in its infancy although major challenges exist (cf. Fig. 1.4). These include (i) the elicitation of requirements, (ii) the analysis of a set of requirements with respect to completeness and consistency, and (iii) the traceability of requirements to the subsequent phases (i.e., implementation as well as testing and debugging). Thus, the goal of the requirements phase is to achieve a formal specification of the requirements based on the source and target metamodels. Consequently, requirements have to be documented, measurable, and on a level of detail to be able to design the system [21].

Challenge 1: Elicitation of Requirements. A major challenge in the requirements phase is the elicitation of them. In a model-to-model transformation scenario the main task of the elicitation phase is to establish a coarse-grained correspondence model on basis of the source and target metamodels. Nevertheless, as emphasized in [58], model transformations are mainly developed ad-hoc, i.e., the analysis phase in model transformation development has been ne-

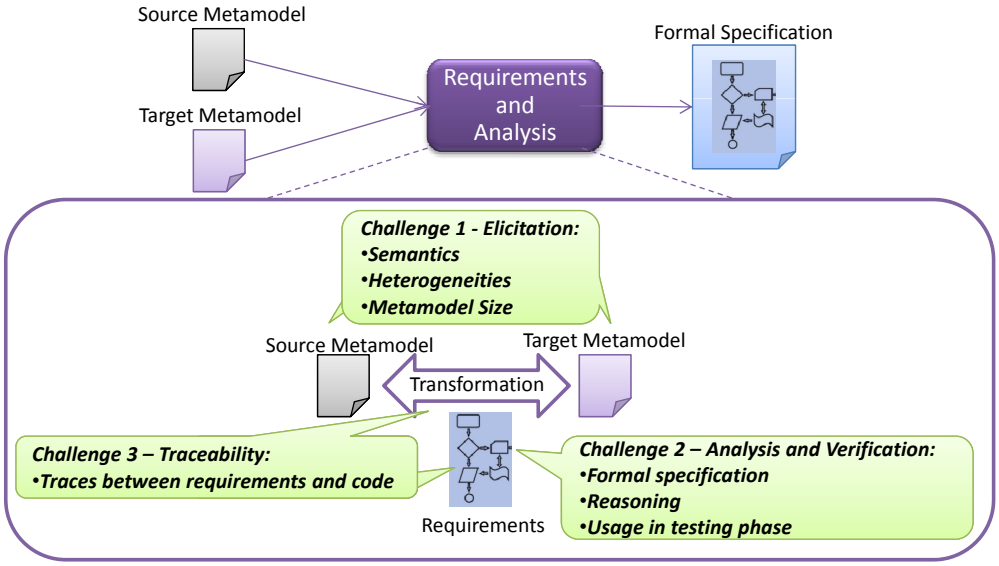


Figure 1.4: Challenges in Requirements Phase

glected so far. However, in order to define the requirements on a transformation (i) semantic and (ii) syntactic heterogeneities between metamodels have to be analyzed.

To enable the realization of a model transformation, first the semantics of the metamodels involved must be understood, such that overlaps in semantics of concepts might be identified. For this, ideally a domain expert is available, who knows about the semantics of the metamodels or corresponding metainformation, e.g., in the form of documentation. Otherwise, this knowledge may be statistically re-engineered from example instances as proposed in the area of data engineering (cf., e.g., [69]). If semantically equivalent concepts have been found syntactic heterogeneities have to be analyzed. Syntactic heterogeneities typically result from the fact that semantically equivalent or related concepts can be expressed by different metamodeling concepts [94]. Nevertheless, in in this context a major challenge is to describe the occurring heterogeneities, since no common vocabulary for heterogeneities has been established so far. For establishing such a common vocabulary, a starting point would be to analyze which heterogeneities might occur at all between metamodels, as done in [86].

Furthermore, due to the increasing complexity of systems and modeling languages, also the size of metamodels grows and thus, mechanisms are needed to analyze large metamodels. To support the transformation designer in this task, *matching tools* might be employed to enhance understanding of large metamodels. This is since matching tools allow to derive correspondences between metamodels which might help to reveal semantic relationships automatically. In this respect, first metamodel matching tools [42, 43, 123] inspired from schema matching [122] have been developed.

Challenge 2: Analysis and Verification of Requirements. Given the fact that a set of requirements has been posed, a major question is whether this set is complete and consistent. Thus, the language that has been used to capture the requirements should enable reasoning on

them with respect to (i) metamodel coverage, (ii) redundancies, and (iii) contradictions. To allow for automatic testing of the requirements, they have to be formally defined [143]. Currently, for the specification of requirements, OCL-based approaches (cf., e.g., [29,30]) have been proposed. Nevertheless, they do not allow for reasoning and are difficult to write and yield to verbose specifications. Thus, in order to facilitate the specification of requirements, a more user-friendly, but still formal language is needed, which is specific to the domain of model transformations. This is in contrast to software engineering, where formal methods like Z [143] or Alloy [70] have been developed. However, such languages are specific to the domain of software engineering in the way that they are, e.g., not capable to deal with the complex structures of models, which makes the specification of requirements tedious. Thus, a dedicated language is needed that takes the special requirements of model transformations into account.

Challenge 3: Traceability of Requirements. Since the recorded requirements serve as contracts for the implementation, traceability of each requirement to the succeeding phases is indispensable. This is also favorable to locate the area of the implementation that has violated a certain requirement as checked in the testing phase. Thus, a mechanism is needed that allows to manage links between requirements and their realization in the transformation code. Such a mechanism could be a simple model allowing to store links between requirements and transformation rules.

In summary, although first approaches supporting the transformation designer in the requirements and analysis phase have been proposed, a dedicated support to formally specify the requirements, which can subsequently be used for testing the specified transformation, is still missing.

Deficiency 1: Missing Support for Specifying Transformation Requirements. In software engineering, dedicated languages have been proposed to capture the requirements, e.g., Z [143] and Alloy [70]. Current transformation languages do not provide means to specify certain requirements. Instead, the requirements a transformation is supposed to fulfill are only available in an informal way, e.g., in terms of a textual description. Furthermore, in software testing, a so-called *oracle* determines if the result of a test case is correct [13, 19], i.e., if no differences between the generated and the expected results exist, the test run succeeds. Such oracles may also be employed in the domain of model transformations to determine if the transformation specification fulfills the posed requirements, e.g., comparing if a generated target model is equal to desired one (cf below). Since the expected target model is not available, usually a so-called *partial oracle* may be employed to check if expected properties hold for the generated model. Concerning the running example of Section 1.2, if there is a `Package` in the source model there should be an according `Schema` in the target model according to the above presented requirements. A possibility to realize partial oracles is to use *design by contract* which has been introduced for object-oriented programming languages [105]. Design by contract allows to formalize requirements in terms of contracts, which may be used to test the software, i.e., contracts on methods specify valid input parameters and report an error in case of invalid values. Providing a dedicated language for specifying contracts also for the domain of model transformations (i) would allow the definition of contracts that are not tied to a particular transformation language, i.e., it should be *implementation independent* (which is especially favorable in MDE since no dedicated standard transformation language has emerged in practice so far [38]), and

(ii) equips designers of transformations with the possibility to state explicitly desired properties of a transformation *before* implementation, which could then be used in the testing phase to verify the actual implementation.

1.3.2 Implementation Phase

Research in model engineering so far has been focussing mainly on implementation languages. This is analogous to software engineering, where analysis and design notations came later, when scalability became an issue [58]. Although numerous transformation languages have been proposed (cf. [38] for an overview), some main challenges in implementation are still open. These include (i) the automatic derivation of the transformation specification from the defined requirements, (ii) the provision of reusable components, and (iii) means to explicate the operational semantics of the according transformation language (cf. Fig. 1.5).

Challenge 1: Automatic Derivation. For enhancing development efficiency, automatization techniques may be employed to derive (parts of) the transformation logics. Existing approaches in this direction used matching techniques that try to find the correspondences between metamodels and then interpret these correspondences to automatically derive the transformation code [42, 43, 123]. Furthermore, example-based approaches have been proposed, that match model instances to derive model correspondences which are used to infer metamodel correspondences [146]. These metamodel correspondences are then again used to derive executable transformation code. However, matching-based approaches may produce good results in case of similar metamodels but fail in case of heterogeneous metamodels.

Challenge 2: Reusable Components. In order to minimize the number of failures in a transformation the reuse of existing, already tested, components should be considered. Although

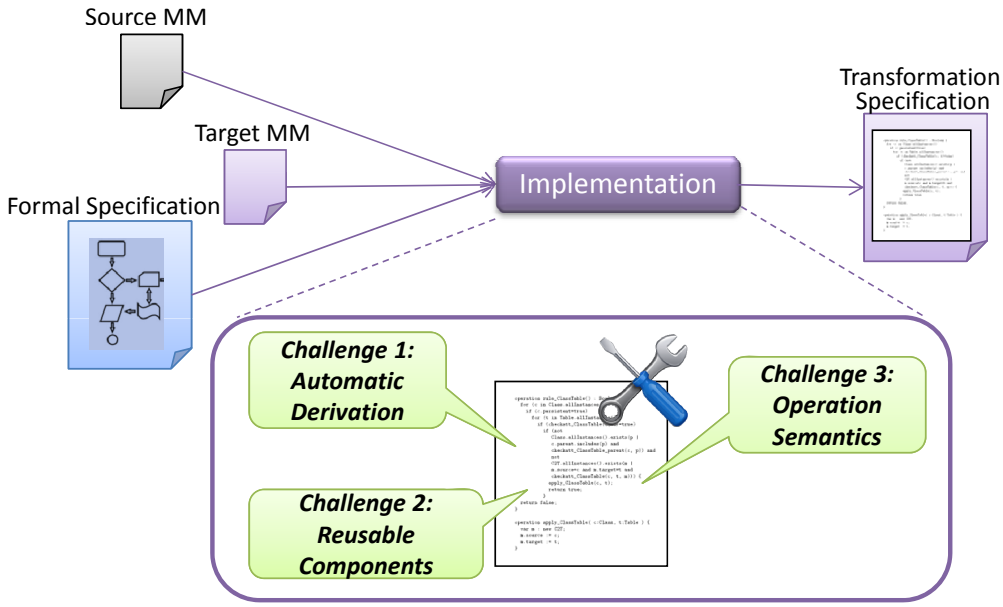


Figure 1.5: Challenges in Implementation Phase

numerous reuse mechanisms (cf., e.g., [37, 42, 159]) have been proposed for model transformation languages it remains unclear in which situations a certain reuse mechanism is suited best. What is missing is an in-depth comparison of proposed reuse mechanisms in rule-based model-to-model transformation languages to highlight when to apply a certain reuse mechanism and how reuse mechanisms complement each other. Such reuse mechanism may be applied to provide reusable transformation components. Finally, inspired from data engineering abstract mappings have been proposed [42, 86] that allow to specify a transformation on a conceptual level by means of high-level components. These components exhibit a well defined operational semantics, which enables the generation of transformation code. Nevertheless, the current proposal does not provide a sophisticated library of components restricting the expressive power. Furthermore, their actual operational semantics is often hard to follow, i.e., it is not clear which component to use in which specific transformation situation.

Challenge 3: Operational Semantics. The diversity of model transformation languages also leads to a diversity in the underlying execution engines, which exhibit different semantics making it hard to comprehend the semantics of a transformation language. Furthermore, current hybrid and declarative model-to-model transformation languages (e.g., ATL [73], TGGs [83], and QVT Relations [116]) specify correspondences between source and target metamodel elements on a high level of abstraction, whereas accompanying execution engines operate on a considerably lower level. For example, ATL uses a stack machine whereas TGGs are first translated to Fujaba storydiagrams [160], which are then again translated to Java for execution. These execution engines act as a black-box to the transformation designer hiding the operational semantics. Furthermore, comprehensibility of transformation logic is further hampered as current transformation languages provide only a limited view on a model transformation problem. For example, in ATL metamodels, models, the transformation specification, and trace information are scattered across different artifacts. Graph transformation approaches using graph patterns only reveal parts of the metamodel. Additionally, both approaches hide the transformation of concrete model elements. The situation is even more aggravated if several tool manufacturers implement a different semantics as is the case for QVT Relational.

In summary, although the focus of current transformation languages is on the implementation phase, further research is needed how to derive the implementation from the findings of the requirements phase (in a model driven way) and how to provide reusable components. Additionally, the diversity of transformation languages and their underlying execution engines, which hide the operational semantics thereof, aggravates the understanding of the actual transformation logic. Additionally this hampers debuggability of model transformations. In this respect, an explicit runtime model, which reveals the internals of the execution engine, is needed.

Deficiency 2: Missing Runtime Model to Investigate the Operational Semantics. If in the testing phase an error occurs, means for understanding and debugging the transformation specification are needed to efficiently find failures. Unfortunately, as discussed above, current transformation engines hide the actual operational semantics. In order to make the hidden operational semantics explicit and consequently following the model-driven approach, model transformations should also be represented as a transformation model as stated in [17]. Although transformation languages base on metamodels, e.g., ATL, only transformation specifications are defined in terms of models but not their execution. If the execution would be represented in

terms of a formal model, the runtime information could easily be used for debugging purposes (cf. below). In this respect, commonalities of different model transformation languages should be represented by such a transformation model which abstracts from technical realization details [17]. Consequently, such a transformation model could not only be used as a conceptual model but could also act as *runtime model* which makes the operational semantics explicit. A runtime model needs to provide means to represent the transformation logic, the metamodels, as well as the respective models involved in a model transformation. It is required to make explicit which model elements are transformed by which transformation rule or due to which circumstances a certain model element may not be transformed, e.g., in case a certain condition is not fulfilled. Additionally, the interconnections between transformation rules need to be made explicit in order to be able to follow the execution order of certain transformation rules which together form the model transformation.

1.3.3 Testing and Debugging Phase

Following the IEEE Standard Glossary of Software Engineering Terminology [68], testing may be seen as a way to verify a system since *testing* is "the process of exercising or evaluating a system by manual or automated means to verify that it satisfies specified requirements, or identify differences between expected and actual results", which has also been discussed in [98]. Baudry et. al stated in [13] that "model transformations constitute a class of programs with unique characteristics that make testing them challenging" whereby this is further aggravated by the complexity of input models and the different transformation languages. According to [64], testing consists of designing test cases, executing the software with those test cases and examining the results produced by those executions. *Model transformation testing* in this sense means that a transformation specification is executed using certain input test models and the generated target models are compared to expected target models. Consequently, methods are needed to verify as to whether a certain source model is correctly transformed into a desired target model. In case a failure is detected, i.e., the transformed model is not equal to the desired target model, means for debugging are required in order to be able to examine the transformation specification and to actually fix the failure.

Following [13], activities in testing of transformations include the generation of test data, the definition of test adequacy criteria to select adequate test cases from the generated test data, and the construction of an oracle, which predicts the expected outcome of a certain transformation. Thus, two main challenges in testing may be described as (i) generating adequate input data and (ii) predicting the outcome for them. Furthermore, to actually decide, whether a problem arises, the predicted outcome must be compared to the actual outcome. Finally, after having recognized that a problem exists, the detection of the failure, i.e., the tracking to the origin of the bug in the code, represents the main third challenge (cf. Fig. 1.6). To actually perform testing and debugging, besides the source and target metamodels as well as the transformation specification also the formal specification may be employed to produce a test protocol.

Challenge 1: Generation of Adequate Input Data. To relieve the tester from the burden of specifying test input models manually, approaches have been proposed for automatically generating valid input models [23, 46, 140]. This is urgently needed due to the complex structure of models, which makes a manual creation tedious and error-prone. In this context, two

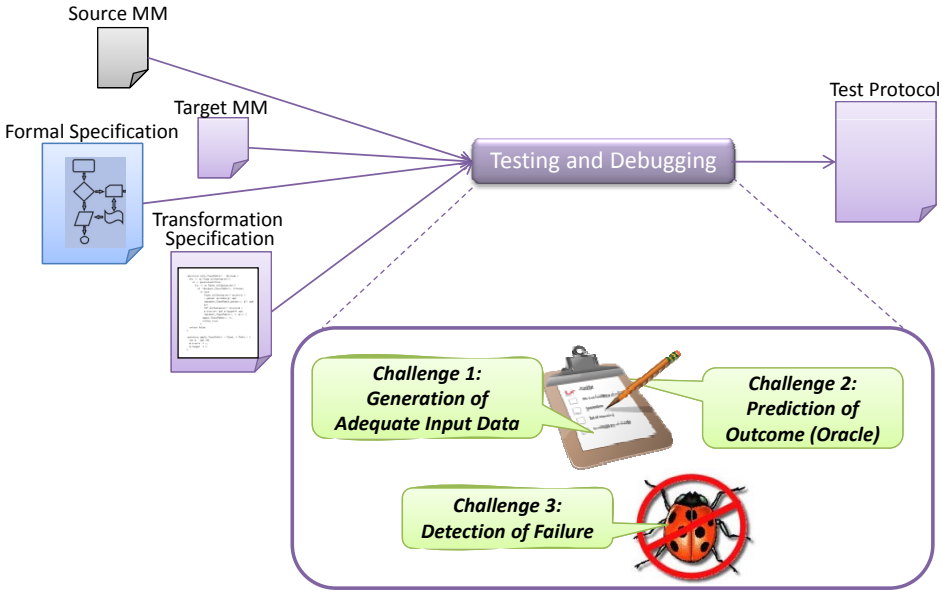


Figure 1.6: Challenges in Testing Phase

main approaches may be pursued. First, input models may be generated based on the source metamodel only. Consequently, a subsequent selection of adequate input models according to existing preconditions must be performed, i.e., the model has to represent a valid subset of models considered by the transformation. Second, input models may be created on the basis of the source metamodel as well as the specified preconditions. In this case, the generation process is more complex, but the subsequent selection process may be omitted. As may be seen from this short overview, several promising approaches concerning the generation of adequate input data have already been brought forward.

Challenge 2: Prediction of Outcome (Oracle). A second major challenge represents the prediction of the outcome for the generated input models, which is commonly denoted as oracle, as already mentioned above. Thereby, complete oracles and partial oracles have been described in literature [13, 109], whereby a complete oracle is responsible to predict complete output models for given input models and a partial oracle predicts only properties that must hold, i.e., if a certain property in the input model holds, a certain property in the output model must be fulfilled. Since in the testing process it should be validated if the afore defined requirements are fulfilled by the transformation specification, it is obvious to reuse the requirements specification for testing. Provided that the requirements have been formally specified, the specification may be made executable in a way that it may serve as partial oracle. Thus, a formal specification of requirements is not only useful in the implementation phase but also in the testing phase. To actually recognize whether failures exist in the implementation, the predicted outcome by the oracle must be compared to the actual outcome of the transformation. Therefore, model comparison techniques are needed to automatically compare an expected output model to a generated output model [98].

Challenge 3: Detection of Failure. Finally, if problems have been observed, the tracking of the origins of failures represents also a major challenge in the testing and debugging phase [173]. Provided that the prediction of the outcome has been performed based on the formal specification of the requirements and that they exhibit traceability means to the implementation, a first hint to the location of a failure may be given. This may then serve as an entry point for debugging. Otherwise, the entry point for debugging must be specified by the transformation designer without any guidance. To enhance the detection of the failure, the provided debugger must make the operational semantics explicit in a way that the impedance mismatch between the transformation specification and the execution engine is kept as low as possible. Another technique would be to apply metrics as proposed in, e.g., [3] to detect so-called code smells.

In summary, although first approaches have been brought forward for the testing and debugging phase, there are still major issues open. These include the prediction of the outcome as well as the detection of the failure.

Deficiency 3: Inappropriate Debugging Facilities. The first major deficiency is to specify requirements on model transformation in way the the may also be used for the testing phase. Nevertheless, this problem has been considered already in Subsection 1.3.1. In order to actually find a failure in a transformation specification, according means for debugging are indispensable. As stated above, most often, current transformation engines are implemented using common object-oriented programming languages like Java, e.g., the stack machine of ATL. As a consequence, debugging of model transformations is limited to the information provided by these programming languages, most often just consisting of variable values and logging messages. Thus, only a snapshot of the actual execution state is provided during debugging while coherence between the specified correspondences is lost. As discussed in [98], a model transformation debugger may not make use of common programming language debuggers "due to the semantic differences in abstraction between the artifacts of code and models". The authors furthermore clearly state that "a model transformation debugger must understand the model representation". In this respect, an execution of a model transformation should again be represented in terms of a model, i.e., an execution should be an instance of the runtime model as described above, to enable debugging on the model representation level. Furthermore, an explicit, model-based representation of the execution state would allow to incorporate more sophisticated debugging facilities known from traditional software engineering, e.g., tracking the origin of a failure by means of reasoning backwards in time and slicing [173]. Finally, if the actual execution is based on commonly agreed, formal methods, properties may be calculated which are useful for debugging. For example, one could reason if the specified transformation terminates or if its behavior is confluent.

1.4 Contributions

The overall goal of this thesis is to provide means to test and debug model transformations. Thereby, the focus is on providing an implementation independent infrastructure to test model transformations against certain requirements and to provide means to foster understandability of model transformations and consequently to ease debugging of model transformations. This thesis therefore presents three major contributions that tackle the aforementioned deficien-

cies of current approaches (cf. Fig. 1.7). First, a declarative, visual language is proposed that allows to specify transformation contracts. Transformation contracts may be used to *observe facts* in model transformation testing according to [173], i.e., to observe what happened in a test run. Nevertheless, further means are needed to understand and debug model transformations, i.e., to *track the origin* and to find the cause of a failure according to [173]. Therefore, the second contribution introduces a common runtime model based on the concepts of Colored Petri Nets (CPNs) [72]. This runtime model builds the basis for the third contribution being sophisticated debugging mechanisms which are especially tailored to the domain of model transformations. In the following the contributions are elaborated in detail.

Contribution 1: Declarative Language to Specify Visual Transformation Contracts. For the specification of partial oracles, a visual, declarative *specification* language to express properties for model-to-model transformation languages is proposed (cf. ① in Fig. 1.7). This language is called PAMOMO (Pattern-based Modelling Language for Model Transformations) and is aimed to allow for the formal definition of model transformation requirements, that may later on be used for testing. First versions of PAMOMO have already been in presented in [55, 57]. In the course of this thesis the language concepts will be modified and extended to make it fit as a contract language. Hence, designers of transformations may use this language to (i) describe desired properties of the transformation, as well as properties of its input and output models in an implementation independent way. This style of properties borrows ideas from the *design by contract* methodology [105] because contracts may be used to specify *preconditions*, *invariants*, and *postconditions* of model transformations. As already mentioned above, preconditions are going beyond metamodel constraints, i.e., they are specific to a certain transformation, and need to be satisfied by input models such that the transformation is applicable. Invariants might be used to specify what conditions need to be satisfied by any pair of input/output models resulting from a correct transformation. Postconditions might be used to express that an output model should or should not contain certain configurations of elements.

One of the advantages of contracts is that they allow to define *what* a piece of software does but not *how* it is done. In this respect, model transformation contracts may be used even before the implementation phase to specify the requirements of model transformations, which are later on used for testing. Additionally, the contract is a useful document for the transformation designer in the development phase, since it describes what the transformation is supposed to do, under which conditions the transformation should be applicable, and which postconditions a transformed model is supposed to fulfill. Thus, the contract makes explicit the requirements of a model transformation to be implemented and may be used to define partial oracles. In this respect, the requirements of the running example in Section 1.2 may be formulated in terms of contracts, i.e., it would be possible to check if a specified transformation fulfills the posed requirements. Consequently, PaMoMo is used for the automated testing of transformation implementations. The specifications are compiled into executable, check-only transformations expressed in the standardized QVT Relations language. These transformations are executed before (to check the preconditions) and after the transformation under test (to check invariants and postconditions) and provide the user with information on which property of the specification was violated (if any) and where. In order to ease the specification and to automate the compilation of specifications, dedicated tool support will be provided.

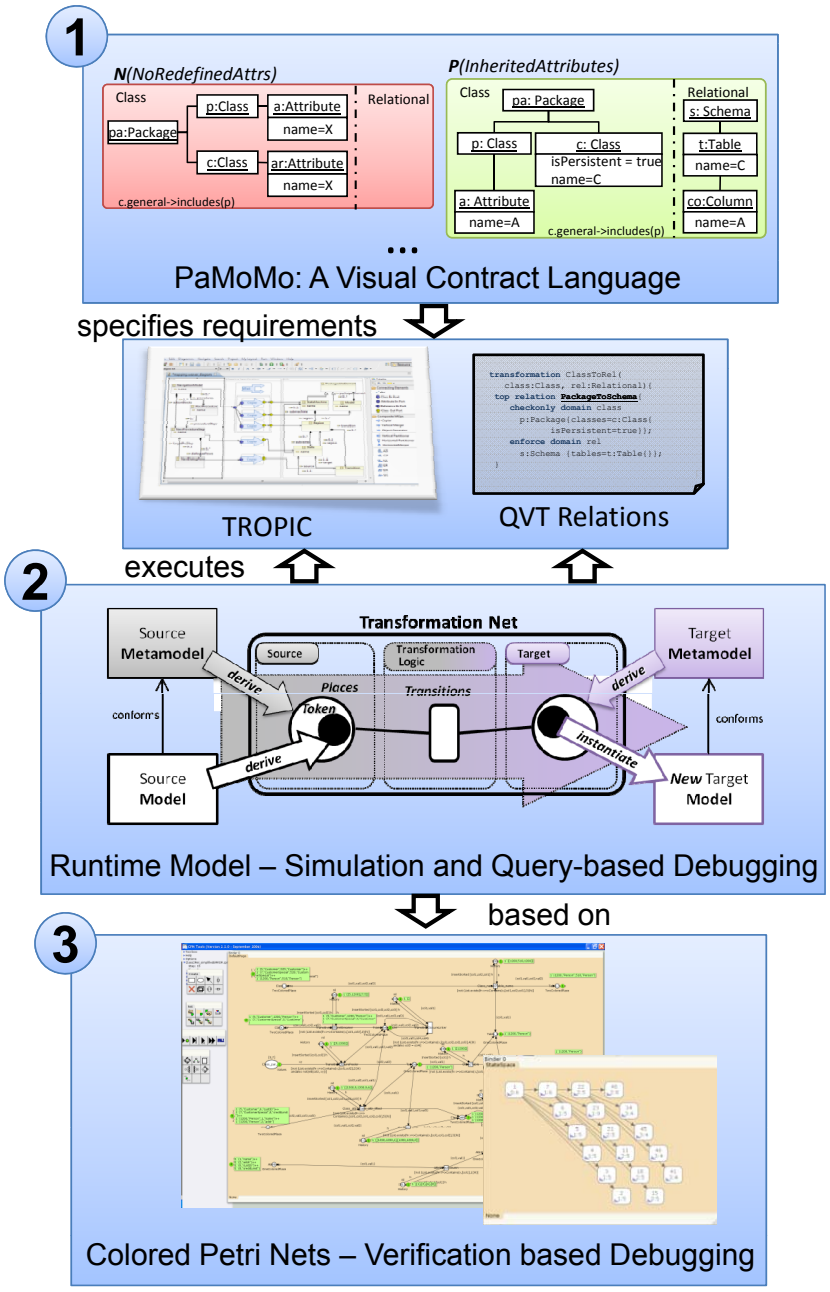


Figure 1.7: Contributions of the Thesis

Contribution 2: A Runtime Model for Model Transformations. Following the idea of model transformations as transformation models [17], this thesis presents Transformation Nets, a DSL on top of Colored Petri Nets (CPNs) [72], for developing, executing, and debugging model transformations (cf. ② in Fig. 1.7). Transformation Nets include commonalities of to-

day's declarative rule-based model-to-model transformation languages. In particular, for every metamodel element, places in Transformation Nets are derived, whereby a corresponding place is created for every class, every attribute and every reference. Model elements are represented by tokens, which are put into the according places. Finally, the actual transformation logic is represented by transitions. The existence of certain model elements, i.e., tokens allows transitions to fire and thus to stream these tokens from source places to target places representing instances of the target metamodel to be created. This approach follows a process-oriented view towards model transformations allowing debugging on an appropriate level of abstraction. Furthermore, Transformation Nets provide the explicit statefulness of imperative approaches through tokens contained within places. The abstraction from control flow known from declarative approaches is achieved as the nets transitions may fire autonomously, thus making use of implicit, data-driven control flow. In this respect, Transformation Nets act as a runtime model for model transformations. A first version of Transformation Nets has already been presented in [125]. Nevertheless, in the course of this thesis further development of the runtime model will be considered, going beyond the contributions proposed in [125]. Thereby, major improvements have been considered in three different directions. First, numerous changes and extension to the initial version of Transformation Nets have been conducted in order to provide a metamodel that represents the commonalities of current rule-based model-to-model transformation languages. Thereby, a focus has been set on representing reuse concepts, e.g., rule inheritance and modularization concepts. Besides changes in the underlying metamodel of Transformation Nets, second, a formal basis has been provided by its full compilation into CPNs. Finally, the compilation into CPNs enables the use of efficient standard execution languages, e.g., CPN Tools[8] in the prototypical implementation, as well as formal properties of CPNs, which may be employed for debugging the transformation specification.

Contribution 3: Debugging Facilities for Model Transformations. The ability to combine all the artifacts involved, i.e., metamodels, models, as well as the actual transformation logic, into a single representation makes the formalism especially suited for gaining an understanding of the internals of a specific model transformation. First, this formalism allows to detect bug-smells by inspecting the static structure of the according Transformation Net. Second, the runtime model serves as a basis to provide debugging facilities, whereby in this thesis three dedicated mechanisms are proposed namely (i) *simulation-based debugging*, (ii) *query-based debugging*, and (iii) *property-based debugging* (cf. ② and ③ in Fig. 1.7). They will shortly be described in the following. As stated in Section 1.3, the runtime model should also include information about the actual execution of a model transformation, i.e., which model elements have already been transformed, which may be used for debugging purposes. This is why Transformation Nets also store the models in terms of tokens. The execution of a model transformation may then be simulated, e.g., by firing transitions in Transformation Nets. The stepwise firing of the transitions makes explicit the operational semantics of the transformation logic and thereby enables *simulation-based debugging*. Furthermore, the runtime model may be exploited by means of *query-based debugging*. For this, OCL queries are proposed which allow to reason backwards in time, e.g., since the execution stack is represented in the model it is possible to query which transition produced a certain target token and which source tokens where involved in creating

[8] http://cpntools.org

a certain target token. Since the execution is stored as a model, *forensic debugging* is enabled additionally, i.e., debugging might occur after having executed the transformation logic based on the explicitly available runtime model. Finally, the formal underpinnings of CPNs allow the application of generally accepted behavioral properties, characterizing the nature of a certain CPN, e.g., to test if a certain target model may be created with the given transformation logic. Thus, CPN properties enable *property-based debugging*.

Prototypical Implementation. Besides discussing the contributions of the thesis from a conceptual point of view, a prototypical implementation of the concepts is provided, which is based on the Eclipse Platform. More specifically, the prototype is based on the Eclipse Modeling Framework (EMF)[9] [24] for specifying the metamodels of PaMoMo and Transformation Nets and on the Graphical Modeling Framework (GMF)[10] [54] for specifying the concrete, graphical syntax of them. To execute PaMoMo contracts they are compiled into checkonly QVT-Relations, more specifically the implementation of Modelmorf[11] is used. CPN Tools are used to execute Transformation Nets as well as to calculate formal properties. To enable communication between the Eclipse based implementation of Transformation Nets and CPN Tools the ASAP platform [163, 164] is used.

1.5 Thesis Outline

The thesis is mainly structured to the well-known development cycle and along the three major contributions depicted in Fig. 1.7. In the following, a brief overview on the structure of the thesis is given.

Chapter 2: Related Work

In order to present the fundamentals of this thesis this chapter presents related work and state of the art in testing and debugging of model transformations. Thereby, the main deficiencies of current approaches are identified which are improved by the contributions of this thesis.

Chapter 3: PaMoMo: A Visual Language for Model Transformation Contracts

This chapter focuses on the introduction of the declarative, visual language PaMoMo (Pattern-based Modelling Language for Model Transformations) which may be used to specify contracts in terms of preconditions, invariants as well as postconditions for model transformations. Finally, it is reported how these contracts may be compiled into QVT-Relations in order to validate the contracts against a certain transformation. The idea of contracts has been summarized in a journal paper which was accepted for publication at the time of writing the thesis [59]

Chapter 4: Transformation Nets - A Runtime Model for Model Transformations

The basic concepts of Transformation Nets, i.e., how metamodel and model as well as the transformation logic itself are represented are introduced in this section. Thus, it is shown that Transformation Nets may serve as a runtime model for declarative model-to-model transformation languages. Parts of the findings in this chapter have been published in several peer reviewed papers [134, 135, 169, 172], initial ideas of Transformation Nets have been published

[9] www.eclipse.org/emf

[10] www.eclipse.org/gmf

[11] http://www.tcs-trddc.com/trddc website/ModelMorf/ModelMorf.htm

in [125, 126, 170]. Furthermore, modularization concepts are discussed, i.e., modules are presented which allow the modularization of Transformation Nets and the definition of reusable components in a transformation specification.

Chapter 5: Rule Inheritance in Transformation Nets

Whereas the previous chapter introduced basic concepts of Transformation Nets, this chapter focuses on reuse mechanisms in Transformation Nets. In this respect, it is discussed how rule inheritance is supported in current transformation languages (published in [167]) and how rule inheritance may be incorporated into Transformation Nets.

Chapter 6: Colored Petri Nets as Semantic Domain for Transformation Nets

Transformation Nets represent a Domain-Specific Language (DSL) on top of Colored-Petri Nets (CPNs) [72]. In order to make use of already existing, efficient CPN execution engines as well as the formal underpinnings, i.e., properties, provided by CPNs, Transformation Nets may be fully compiled to CPNs. This chapter therefore (i) introduces CPNs, (ii) formalizes Transformation Nets according to principles of Petri Nets and presents the (iii) actual compilation.

Chapter 7: Debugging Support for Model Transformations

After translating Transformation Nets to CPNs and thus providing a formal execution engine, this chapter discusses how (i) *simulation-based debugging*, (ii) *query-based debugging*, and (iv) *property-based debugging* are realized. The findings in this chapter are partly published in [135, 165].

Chapter 8: Prototype Implementation

Besides presenting testing and debugging of model transformations from a conceptual point of view, this chapter provides an overview of the actual implementation. In this respect, first, the implementation of PaMoMo is discussed, followed by the realization of Transformation Nets and their according debugging facilities.

Chapter 9: Evaluation

In order to evaluate and to critically reflect the presented contributions this chapter presents evaluations on the basis of case studies and comparative reviews. First, it is shown how a transformation specification defined in QVT Relations may be tested by means of contracts and debugged by means of a translation to Transformations Nets, partly published in [168, 171]. Second, a comparative study driven by the identified deficiencies in related work is conducted in order to show in which way the thesis improved the state of the art. Additionally it is shown how Transformation Nets may be used to specify the operational semantics of the mapping language called Mapping Operators (MOps) [86].

Chapter 10: Conclusion and Future Work

The thesis concludes with a summary and a critical discussion of the achieved contributions. Finally, current limitations as well as an outlook on potential further research directions is given.

Chapter 2

Related Work

Because things are the way they are,
things will not stay the way they are.

— Bertolt Brecht

Contents

After having shortly discussed current deficiencies, this chapter provides an in-depth overview on related work. Thereby, the related approaches are separated into the three main deficiencies identified in Chapter 1. First methods are surveyed concerning testing of model-to-model transformations and how test cases can be specified. The term testing refers to "the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspect of the system or component" [68] in order to detect if *failures* in a program exists. According to [173], a failure is an *infection* that is externally observable. An infection occurs if the *defect* in a program is executed in a way that the desired state of the program differs from the actual state. *Debugging* is then the process of locating defects in a program as well as to remove the defect so that the failure no longer occurs. Therefore, in this thesis the term failure is preferred to the term *bug*, which is not precisely defined [173, p. 19]. In order to find a failure, tool support is indispensable. Second, related work concerning runtime models is presented, which may serve as basis for a debugger in declarative rule-based model-to-model transformation languages. Finally, related work concerning debugging support in current model transformation languages is presented.

2.1 Model Transformation Testing

The need for systematic testing of model-to-model transformations has been recognized by the research community and has been documented by several publications, outlining the challenges to be tackled [12, 13]. As a response, several testing approaches have been proposed whereby Fig. 2.1 shows a classification thereof.

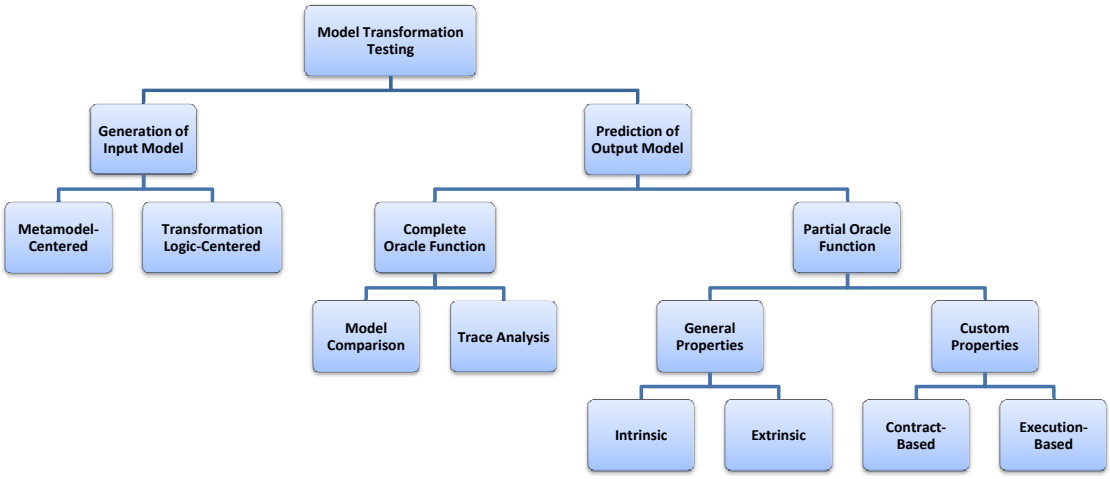


Figure 2.1: Classification of Existing Approaches in Model Transformation Testing

Basically, existing approaches can be divided into approaches (i) for the automated *generation of test input* and (ii) for the *prediction* of the desired *output* model. Considering approaches for the generation of input models, they can be further divided into those that make use of the source metamodel of the transformation only, to systematically generate a large set of test cases or those that additionally take into account the design and implementation of the model transformation. Approaches that solely base on the metamodel are also called *black-box* generation approaches, whereas those making use of the implementation are called *white-box* approaches [13]. Methods that try to predict the desired output model can again be split into those that try to predict the whole output model, i.e., that make use of a *complete oracle function*, and those that try to predict parts of desired target model only, i.e., *partial oracle functions*. Complete oracle functions may be defined by having the expected output model at hand, which acts as a reference model for analyzing the actual output model of a transformation, e.g., by means of *model comparison*. Other approaches use the oracle function to compare an existing trace model with the actually produced trace model (cf. *trace analysis* in Fig. 2.1). Since often no desired target model is available and since it is tedious and error-prone to establish a target model manually, approaches exists that try to ensure certain properties of a transformation by means of *partial oracle functions*. Thereby, approaches consider either testing *general properties* such as confluence, applicability and termination of a set of transformation rules, which are generally applicable to all transformation specifications, or *custom properties* which are specific to a certain transformation specification. Considering general properties, either the transformation language itself provides means to test them, i.e., *intrinsic* in Fig. 2.1, or the transformation

specification is transformed to a specific language that allows to test properties, e.g., Petri Nets or constraint solvers, i.e., *extrinsic* in Fig. 2.1. Finally, custom properties can be specified based on *contracts* or on their *execution*. Contracts allow to specify preconditions, invariants and postconditions that need to be fulfilled by a transformation specification. Execution based approaches allow to specify a formal property that can be automatically checked by analyzing the state space, which contains all possible execution sequences of a transformation specification. This is especially common in model checking. In the following, the classification is explained in detail and related work in testing model transformations will be presented.

2.1.1 Automated Generation of Test Input Models

As stated, e.g., in [23], test data generation for model transformations requires to handle complex data structures compared to the data of traditional programming. This difference requires special means for test data generation, i.e., how to generate source input models. Due to this complexity, the manual specification of input test models would be tedious and error prone. Therefore, automated generation is heavily needed. A distinguishing criteria amongst the approaches presented is, as to weather the approach makes use of the metamodel only, or additionally considers the actual transformation specification under test.

Metamodel-Centered Approaches. An approach that uses the metamodel to generate test input models is presented in [23]. The presented algorithm first derives so-called model fragments which specify parts of the metamodel that should be instantiated with interesting values for testing, e.g., minimum values, maximum values, or null values. The presented approach then combines and completes the fragments in a way that a valid source model results, i.e., the source model has to conform to its according metamodel. The focus of the paper is to present different strategies how to achieve this goal. In a subsequent work the authors present a framework to qualify the relevance of generated input model for testing [45]. Furthermore, the framework identifies missing model elements in input models and assists the user in improving these models. Another point that is highlighted in [140] is that a metamodel might exhibit additional constraints, e.g., by means of OCL constraints. These constraints have to be fulfilled by the generated input models. Therefore, an automatic approach is presented which is based on constraint satisfaction. Based on previous work of the authors presented in [139], the input metamodel and its additional constraints are translated to Alloy[1] to generate a boolean formula which is solved using a SAT solver [107] to obtain a potential solution.

Transformation Logic-Centered Approaches. The following approaches take into account the design and the implementation of the model transformation for constructing test cases. An approach that considers the matching phase of graph transformation rules to test input models is presented in [39]. Based on a fault model for graph transformations, the extracted model that originally matches a certain graph transformation rule is systematically changed in order to introduce errors which have to be detected during testing. In [87], a template language to generate test models based on the structure of the rules used to implement the transformation is presented. A given transformation rule can be transformed into a so-called metamodel tem-

[1] http://alloy.mit.edu/community

plate. Such metamodel templates are then used to automatically create template instances that represent suitable test cases.

In summary, it has to be emphasized that the generation of test input models is not the focus of this thesis. Nevertheless, the generation of input models has been discussed shortly to (i) provide a full overview on related work concerning model transformation testing and (ii) because the methods that will be presented throughout this thesis require according input models, which may be generated using the above presented approaches.

2.1.2 Prediction of Output

In general, literature on model transformation testing distinguishes between two kinds of oracle functions [13, 109] to predict the output of a model transformation. First, *complete oracle functions* may be defined by providing a full-fledged expected output model for each test input model, and subsequently, employing *model comparison* frameworks to verify the equality of the actual output model with the expected target model. Second, *partial oracle functions* can be used to test the transformation specification against desired properties, i.e., if the generated target model contains a certain graph structure.

2.1.2.1 Complete Oracle

Verification by Model Comparison. Complete oracle functions may be defined by having the expected output model at hand which acts as a reference model for analyzing the actual output model of a transformation as proposed in [82, 97, 98]. Model comparison frameworks are employed for computing a difference model between the expected and the actual output models. If there are differences then there is an error. However, reasoning about the cause for the mismatch solely bases on the difference model (comprising differences such as additions, deletions, movements, and updates of model elements) is challenging. The situation is even more aggravated since several elements in the difference model may be caused by the same error. The transformation designer has the burden to cluster the differences by himself. For large test input models which result in large output models, this approach seems unfeasible in practice, and partial oracle functions are more appropriate.

Trace Analysis. Instead of applying an oracle to compare the generated target to a desired one, in [78] an approach is presented to specify oracle functions solely on the basis of *trace links* between the input models and the output models. Thereby, the authors propose an "oracle function that compares test cases with a base of examples of existing traces". This leads to the fact, that no expected target model needs to be defined for every test case. Additionally, the trace links may act as pointers to the location of a potential error. Nevertheless, this approach assumes the existence of an initial trace as a basis for the oracle function, which is hardly the case in practice.

In summary, complete oracle functions require the execution of the actual transformation under test to receive the target model or the trace links required for comparison. This execution based testing has several advantages, as stated in [97], i.e., (i) it is easy to perform the actual testing (compared to more formal methods presented in the following), (ii) the specified model transformation is executed in its expected environment, and finally (iii) the testing process can

be automated, i.e., it is possible to generate the input models, execute transformation and do the comparison without user interaction. Nonetheless, often no desired target model is available for model comparison and reasoning about the actual failure is cumbersome since only difference models are available. To overcome these limitations, partial oracle functions have been proposed which are discussed in the following.

2.1.2.2 Partial Oracle

Partial oracle functions have been proposed for checking properties of input models, output models, and their relationships. Thereby, properties which are applicable to transformations in general may be specified, i.e., properties such as termination or confluence, but also properties specific to a single transformation, e.g., a property to check if "for every persistent class an according table has been created" as desired in the running example.

General Properties. Several properties that are common to all model transformations may be checked. For example, every model-to-model transformation has to be finite, i.e., the model transformation has to terminate. Additionally, a transformation should not run into a deadlock. Furthermore, it is desirable that a model-to-model transformation is confluent, i.e., the produced target model should always be the same, provided that the same input model is used. Several approaches have been presented to verify such common properties. Those are detailed in the following. Thereby, existing approaches can further be divided into *intrinsic* and *extrinsic* approaches.

Intrinsic: A transformation language that provides support for checking general properties is AGG [147]. Nevertheless, the proposed approach of analyzing *critical pairs* is applicable to any graph rewriting system and could therefore be included in other graph transformation languages as well. In this respect, Heckel et. al [65] showed how critical pair analysis can be applied to check for confluence of a specified transformation. A critical pair in graph transformation rules occurs if two transformation rules are non-parallel independent. Parallel independence means that "two rules can be applied in any order yielding the same result. Otherwise, if one of two alternatives is not independent of the second, the second will disable the first. In this case, the two steps are in conflict" [65, p. 8]. Formal proofs are provided which show that if critical pairs are confluent then also the transformation is locally confluent.

Extrinsic: The term extrinsic partial oracle means that the model transformation language itself does not provide means to test certain properties, but instead the properties are tested in some external formalism. Especially in the area of graph transformations, work has been conducted that use Petri Nets to check formal properties of graph production rules. The approach proposed in [157] translates individual graph rules into a Place/Transition Net and checks for its termination, since termination is in general undecidable in graph grammars [119]. Another approach is described in [90], where the operational semantics of a visual language in the domain of production systems is described with graph transformations. The models of the production system, as well as the graph transformation rules are transformed into Petri Nets in order to make use of the formal verification techniques for checking properties of the production system models. Thereby a short discussion is given, how the properties for Petri Nets presented in [112] may be applied in the domain of model transformations. Nevertheless, these approaches make some abstractions, i.e., the derived Petri Nets abstract from details of the model transformation.

For the approach presented in [157], it is stated that the derived Place/Transition Net is only a simulation of the specified transformation rules, but not a bisimulation. Instead of Petri Nets, the approach presented in [27] uses OCL to verify common properties of a transformation. Thereby, the graph transformation rules are represented by an intermediate OCL representation which allows to test certain properties with OCL invariants. In the paper seven general properties have been presented, for example, applicability, i.e., if a certain rule is applicable at least once, or conflict, i.e., two rules are in conflict if firing one rule can disable the other one.

Custom Properties. In addition to general properties, often transformation specific properties have to be checked in order to test the correctness of a specific model-to-model transformation. Mainly two approaches can be distinguished, namely contract based approaches and approaches that make use of model checking techniques. In the following, representative papers are presented for both categories.

Contract-Based Approaches: Contracts are a well-established technique in software engineering to verify object-oriented programs [93, 105]. Inspired from these ideas, contracts have also been applied for the verification of model transformations in previous research. In the following, several approaches proposed for verifying model transformations using contracts are discussed, divided into (i) OCL based, (ii) graph pattern based, and (iii) model-fragment based approaches.

- *OCL Based Approaches:* The first approach using contracts for model transformations was proposed by Cariou et al. [29, 30]. The authors suggest implementing transformations with OCL. In this way, the source metamodel classes are provided with operations, which may comprise preconditions, postconditions, and invariants. Although OCL natively supports design-by-contract, OCL is not intended to specify transformations and relationships between models. Thus, the authors propose an extension for OCL that allows defining mappings between input and output model elements. A similar approach for defining contracts with OCL has been proposed in [108]. Besides other aspects, Kuester et al. [87] also agree on the use of OCL for the definition of transformation specific constraints for the produced output models. Common to all these approaches is that the contracts are embedded into the underlying transformation during execution. In this respect, neither the specified preconditions nor the postconditions can be checked without the execution of the transformation. Furthermore, a tool is needed that allows to execute the transformation and to check the constraints at the same time, which reduces applicability, as stated in [29].

  Additionally, Cabot et. al showed in several papers, how OCL invariants can be derived from graph transformations specified in TGGs or QVT Relations [25, 26, 28]. In this respect, the invariants state conditions for a valid transformation. The derived invariants together with the source and target metamodels form the so-called transformation model. The transformation model is then used to test the specified model transformation using the author's UMLtoCSP tool [28] for analysis. Thereby the invariants are translated to a constraint solving problem and according constraint solver can thus prove the properties. Furthermore, UMLtoCSP is able to generate a valid combination of source and target model, which may be used to check if the generated model are equal to a desired one (cf.

above). In [52], a mechanism is presented to define properties for source models, target models, and source-target relationships as contracts expressed in OCL based on transformation contracts. A transformation contract defines a set of properties together with a set of valid source models. These source models are then transformed by the transformation under test into according target models, whereby the resulting target models are checked by the USE tool [2], which evaluates the specified OCL constraints.

In [81], the authors propose the Epsilon Unit Testing Language to test model management operations. The language permits defining test operations where post-conditions for the model transformation under test may be specified. In a similar vein, Giner and Pelechano [51] propose a Test-Driven approach to the construction of model transformations. Their focus is to capture the requirements for a model transformations by means that can be later on used for testing, i.e., the requirements are covered in the form of test cases made of an input model, together with output fragments and OCL assertions. These test cases act as contracts for the to be specified transformation, which can be tested against those requirements.

In [55] and [58] TRANSML as language to cover the life-cycle of transformation development enabling the engineering of transformations has been presented. TRANSML includes a dedicated language for model-based testing, which enables the description of test cases. Thereby, the expected properties may be described in an OCL like textual syntax. The specified test cases can be executed after the according transformation under test in order to check if the specified properties are fulfilled.

- *Graph Pattern Based Approaches:* In [8], the authors propose to use the patterns supported by the VIATRA2 tool to specify contracts for model transformations. However, their patterns operate on one metamodel only, being therefore usable to specify pre- and postconditions, but not transformation invariants.

- *Model-Fragment Based Approaches:* Finally, a special form of contracts was presented in [109]. Based on the findings in [123], the authors propose to use model fragments for defining properties which are expected for an output model produced from a specific input model. For verifying these properties, the model fragments are matched on the produced output model. This approach is different from the previous ones, which propose using generic contracts solely defined on the metamodel level and not specific to a concrete test input model. The advantage of using model fragments is to support a user-friendly specification of test cases by reusing the graphical modeling editors, but this induces that the constraints are described at the model level. Thus, they have to be defined for each particular test input model.

Execution-Based Approaches: In order to verify the actual execution of a model transformation, techniques from model checking are employed to test model transformations. Model checking [36] is a method to check a model against a certain specification. Typically the state space of the system is calculated which is used for verifying the specification, usually expressed

[2] http://www.db.informatik.uni-bremen.de/projects/USE

in terms of temporal logic formula, i.e., a property that may be checked in the course of the running example might be that there exists a state in the future where there are exactly as many tables as classes available. In this respect, in [156] a translation of graph transformation rules to transition systems, serving as the mathematical formalism of various different model checkers, has been proposed. Thereby, only the dynamic parts of the graph transformation systems are transformed to the transition system in order to reduce the state space. The GROOVE toolkit[3] provides model checking facilities based on graph transformations [128]. Their main goal is to enable verification of object-oriented programs where the behavioral semantics is expressed by means of graph grammars. From the graph grammars an according transition system is generated which allows to check properties expressed in a temporal logic on graphs [127]. In Rivera et. al [129], the transformation from graph transformations into the rewriting-logic based language Maude[4] is presented, which provides explicit means for formal analysis. Consequently, model checking and reachability analysis is enabled for graph transformations. A similar approach is presented in [10], whereby Alloy [70] is used as a model checking language. Nevertheless, none of the presented approaches states how the results of the verification of the properties can be mapped back to the actual transformation language. In this respect, the upcoming implementation of the graph transformation language Henshin[5] provides direct support for state-space analysis and tries to integrate this technique into the transformation language. OCL can be employed to check for properties on the state space. Nevertheless, the analysis method are restricted to in-place transformations only.

In summary, testing model transformations by means of partial oracles is a promising approach as the plethora of related work reveals. However, the ones based on OCL usually lead to complex constraints, which are difficult to write in practice, yielding verbose specifications [30], especially for the specification of relations between input and output models. Furthermore, the specification of the partial oracles is tightly coupled with the actual transformation language. This means that every transformation language requires specific methods to specify contracts or to translate the transformation specification into a formal language. Finally, by using a pure OCL-based approach, only true or false is given back as answer to the user, but no further information is accessible in standard OCL environments. Approaches based on model checking typically provide a counter example that caused the property to fail. Nevertheless, this information is represented by means of internal states of the model checking tools only, whereas the mapping of the results back to the according transformation language is left open.

2.2 Runtime Models for Model Transformations

In general, run-time models provide means to abstract from code level details to allow for reasoning on the runtime behavior of a system [95]. Up to now, runtime models for model transformations have not gained much attention. In [17], it was presented that the specification of model transformations itself should be model-based. In this respect, a metamodel for a model transformation language has been proposed. The explicit representation allows to use the

[3] http://groove.sourceforge.net

[4] http://maude.cs.uiuc.edu

[5] http://www.eclipse.org/modeling/emft/henshin

specified transformation similar to any other model. In particular a model transformation may again act as an input for a model transformation, which is then called a higher-order transformation [152]. Nevertheless, the proposed metamodel does not include any information on the actual execution of a model transformation. Only the approach presented in [88] provides a first step towards this direction. Thereby, they authors made explicit the operational semantics of QVT Relations by representing them in terms of CPNs, allowing the transformation designer to conclude about the operational semantics of QVT Relations. Finally, ATL [73] explicitly represents the transformation specification as a model, which is then compiled into a virtual machine. In this respect, the underlying virtual machine could be seen as runtime model, but acting on a very low-level of abstraction.

Considering software engineering in general, proposals exists that try to provide runtime models on the actual execution of a system. These models are then used for *runtime verification* [11]. The main idea is to extract information from a running system in order to check if the observed behavior satisfies or violates certain properties. In this respect, runtime verification builds the basis for many purposes, such as monitoring or debugging. Runtime verification tries to avoid the drawbacks of formal verification techniques such as model checking, i.e., the systems needs not to be formally modeled and there is no need for the exhaustive calculation of the state space. Nevertheless, it does not provide full coverage. Work in this area has been conducted by Nierstrasz et al. [113], focusing on providing according abstractions and visualizations of the runtime behavior of a system. In [32], a framework has been presented which allows a developer to specify a certain property that should be checked by using runtime verification. In this case the properties are injected into the code using techniques from aspect-oriented programming. If a property fails, the according trace, i.e., which statements have been executed so far, is provided to the programmer, which allows to check for failures. In this respect, Maoz suggested to use model-based traces for runtime verification [102]. The runtime information provided by means of standardized trace models can then be easily presented to the programmer, e.g., in the form of sequence diagrams, statecharts or class diagrams. Maoz also proposed metrics and operators, i.e., filters to hide certain parts of the execution or comparators to compare two executions of the systems, to analyze the provided trace information. Furthermore, dependency graphs may be built which can be used to realize dynamic slicing [173], allowing to reason about the execution of the system.

In summary, for model transformation currently no dedicated runtime models exists that would allow the transformation designer to reason on the execution of the specified transformation. Nonetheless, a runtime model could be employed for runtime verification techniques of model transformations and could be used for debugging model transformations as well.

2.3 Debugging of Model Transformations

In order to find a failure in software engineering, a common feature of every integrated development environment is to allow the programmer to monitor and potentially alter the state of a running programm by means of a debugger. Nevertheless, debugging support for model transformations is still in its infancy [85]. As stated in [101], the miss of appropriate debugging facilities in the domain of MDE in general and in model transformations in particular hinders

the adoption of MDE in industry. Thereby, debugging on an appropriate level is of utmost importance for declarative (transformation) languages as stated in [158]. This is since declarative languages typically abstract from *how* something is done and the hidden operational semantics of the transformation engine is counterproductive for debugging. Consequently, during debugging exactly this hidden operational semantics needs to be made explicit. First approaches which provide debugging support for model transformations focus mainly on providing runtime information from the actual underlying low-level execution engine, which is often written in a common programming language, e.g., Java. However, the information provided consists of low-level information only, e.g., only variable values are presented to the transformation designer. Therefore, special debugging support is required for declarative transformation languages. In the following, relevant criteria for debuggers of declarative model transformation languages are presented, which are used to compare the debugging support of current transformation languages later in the thesis.

2.3.1 Comparison Criteria

The common goal of debugging is to ease the localization of failures. A debugger should support the transformation designer in narrowing the potential set of causes to a minimum [137]. Based on the requirements defined in [173] for debugging in general and in [101] for debugging domain-specific modeling languages in particular, criteria for debugging of declarative model-to-model transformation languages (or the declarative parts in case of hybrid languages) are derived. Debugging can be divided into *live debugging*, i.e., the transformation is investigated during execution, and *forensic debugging*, i.e., the trace information (which source element has been transformed to which target element) is analyzed to reason on potential failures, as can be seen in Fig. 2.2. Live debugging can further be divided into means (i) to support the transformation designer in *selecting* a certain part of the transformation specification, (ii) to allow the transformation designer to *investigate* the actual state of execution, (iii) to be able to investigate the *dynamic behavior* and finally, (iv) to allow for *adaptations* during debugging. In the following the criteria are explained in detail.

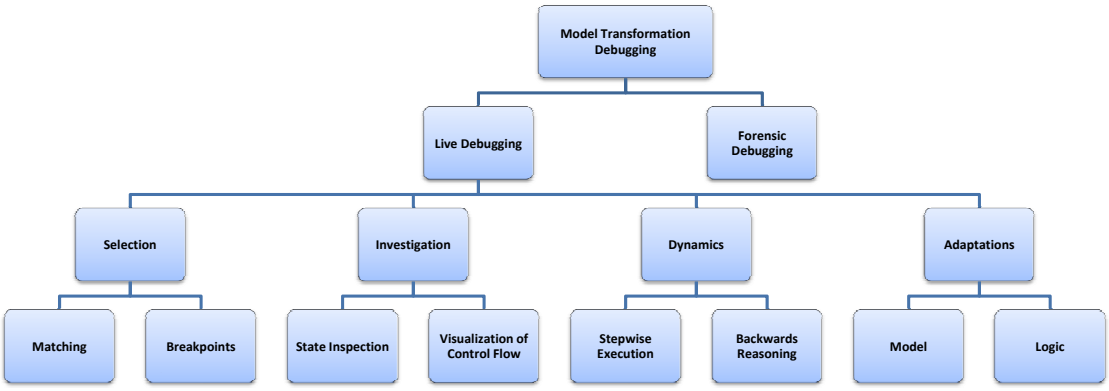


Figure 2.2: Classification of Means for Debugging in Model Transformations

2.3.1.1 Selection

Debuggers need to provide means allowing the transformation designer to select a certain part of the transformation code for execution. Thereby, (i) the matching of models elements should be influenceable by the transformation designer and (ii) breakpoints should be provided to stop the execution at a certain point of time.

Visualization of Matching Process. An important step when executing model transformations is the *matching process*. In general, the matching of elements of the source model and thus the potential application of rules in declarative model transformation languages involves non-determinism. In this respect, the potential choices for matching should be accordingly presented to the transformation designer, i.e., it should be clearly pointed out *which rule* might be executed. Furthermore, it should be possible to influence the choice of the rule application, i.e., the transformation designer should be able to chose the rule. Moreover, a rule might match for several model objects. Consequently, it should be possible to select a certain configuration of model elements, i.e. *what binding* should be used to execute a rule.

Breakpoints. Breakpoints are a common concept in debuggers, indicating that the execution should be stopped before the execution of a certain statement. Nevertheless, the situation in declarative transformation languages offers again certain possibilities, i.e., if the breakpoint is put on a rule, it could either stop the execution before a certain configuration of model elements is matched, only if a certain configuration of model elements was successfully matched or even in case a certain configuration failed to match. All of these three scenarios could be beneficial in certain situations, i.e., the breakpoint should be configurable. Another possibility of configuration is the specification of *conditional breakpoints* [173], i.e., the breakpoint stops the execution only if a certain user-defined condition is fulfilled.

2.3.1.2 Investigation

In order to reason about the state of execution a transformation designer must be enabled to inspect the current execution state. Additionally, the actual control flow should be visualized within the transformation specification.

State Inspection. Debuggers in traditional programming languages provide means to visualize the current execution state, i.e., typically a view on the memory is provided, representing the actual values of variables. Nevertheless, not only the state of the execution engine should be presented to the transformation designer but also the state of the rules and the trace model, e.g., which model elements have already been transformed by a certain rule, as well as the state of the source and the target model, e.g., which target elements have already been created.

Visualization of Control Flow. The actual state of execution should also be shown in the according code, e.g., in textual model transformation languages the according line of code is highlighted or in graphical transformation languages the according rule may be highlighted. Furthermore, not only the according rule but also the involved model elements should be visualized, e.g., matched model elements in the source model.

2.3.1.3 Dynamics

To allow the transformation designer to reason about the semantics, stepwise execution needs to be enabled. Additionally, a failure is often only detected after executing a certain piece of code. Thus, in order to find the origin of a failure it should be possible to reason on previous execution states.

Stepwise Execution. In order to debug the transformation specification, a stepwise execution should be enabled in order to gain an insight into the operational semantics. Nevertheless, as stated in [137], a notion of an execution step is missing for declarative transformation languages. This is in contrast to imperative transformation languages where instructions are typically the smallest unit of execution which serve as according execution step in debugging. In declarative programming languages the underlying execution engine might execute several actions to perform a certain step, e.g., in order to match for model elements several steps in the underlying execution engine may be required. Consequently, these abstraction from underlying details should also be hidden during debugging, i.e., the transformation designer is not necessarily interested in how a certain model element is matched but only why it can or can not be matched by a certain transformation rule.

Backwards Reasoning. During debugging it should not only be possible to execute the transformation in a forward direction, but it should also be possible to reason about a previous state, i.e., it should be possible to detected failures by reasoning backwards in time to find the origin of the failure [96,173], being closely related to omniscient debugging [120], i.e., recording a program's state over time, and program slicing [150]. In this respect, recent work by Ujhelyi et al. [153] presented a dynamic backward slicing approach model transformations based on automatically generated execution trace models of transformations.

2.3.1.4 Adaptations

Debuggers in common programming languages not only allow a programmer to inspect the execution state but also to manipulate the execution state during debugging, i.e., values of variables might be manipulated or even the code itself might be adapted. In this respect, a debugger for transformation languages should allow the transformation designer (i) to change the model, i.e., to add, edit and delete model elements, and (ii) the transformation logic, i.e., it should be possible to fix failures in the transformation logic within the debugger.

2.3.1.5 Forensic Debugging.

Hibberd et al. [66] present forensic debugging techniques for model transformations by utilizing the trace information of model transformation executions for determining the relationships between source elements, target elements, and the involved transformation logic implemented in Tefkat [91]. With the help of such trace information, it is possible to answer debugging questions implemented as queries which are important for localizing failures. In addition, they present a technique based on program slicing [162, 173], i.e., to identify only those path of a programm that influence a certain state, for further narrowing the area where a failure might be located. Such techniques are beneficial in addition to the debugging techniques described above.

2.3.2 Comparison of Debugging Support in Transformation Languages

Based on the above presented criteria in the following, the debugging support of ATL [73], AGG [147], Fujaba [160], GReAT [6], TGG [83, 136] and QVT Relations using the implementation of mediniQVT[6] are compared, as summarized in Table 2.1.

Table 2.1: Comparison of Debugging Support in Declarative Model Transformation Languages

	Live Debugging									Forensic Debugging
	Selection			Investigation		Dynamics		Adaptations		
	Matching	Breakpoints		State inspection	Visualization of Control Flow	Stepwise Execution	Backwards Reasoning	Model	Logic	
		simple	conditional							
ATL	×	✓	×	✓	✓	✓	×	×	×	×
AGG	✓	×	×	✓	✓	✓	×	~	~	×
Fujaba	×	✓	~ (proposed)	✓	✓	✓	~ (proposed)	~ (proposed)	×	×
GReAT	×	✓	×	✓	✓	✓	×	×	×	×
TGG	~ (proposed)	~ (proposed)	~ (proposed)	~ (proposed)	~ (proposed)	~ (proposed)	×	×	×	~ (proposed)
mediniQVT	×	✓	✓	✓	✓	✓	×	✓	×	×

ATL: ATL uses a stack machine implemented in Java to interpret the transformation specification. In this respect, the ATL debugger is based on the Eclipse Java debugging environment. Since ATL does not make the involved models explicit, the transformation designer is not able to influence the matching process. Concerning breakpoints, ATL supports simple breakpoints only (which have to be specified in the Eclipse outline view). The information about the actual execution state is restricted to low-level information only, i.e., the actual values of the variables are presented to the user (cf. Fig. 2.3). Thereby, the bound objects are presented in the variables view of the Eclipse debugging environment. Nevertheless, the structure of the model gets lost, i.e., the graph like structure is not represented. The according lines of code are highlighted in order to visualize the control flow. A stepwise debugging is possible, i.e., the user is able to check the evaluation of conditions or certain assignments, but only in a forward direction. Furthermore, the values of the variables can only be inspected but cannot be changed during

[6] www.ikv.de

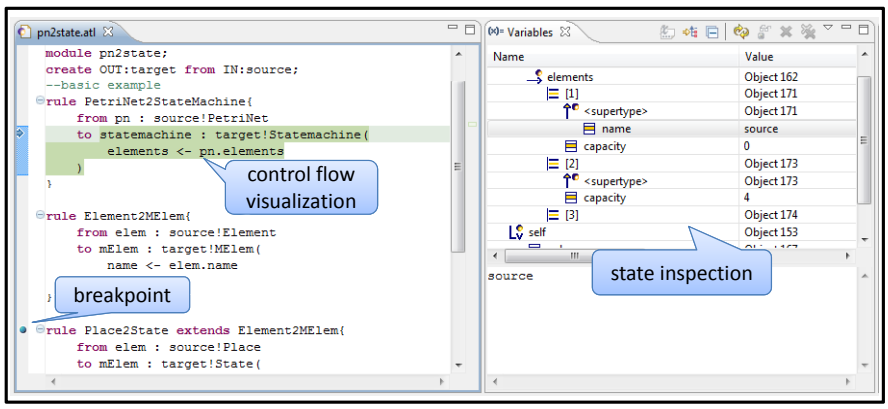


Figure 2.3: Screenshot of ATL Debugger.

debugging. Since the ATL execution engine maintains the trace model internally and during the actual execution only, i.e., it is not explicitly shown to the transformation designer, no forensic debugging is supported.

AGG: Although AGG does not explicitly provide a debugger, it has nevertheless been considered in the comparison, since the execution of AGG allows for (i) an *interpretation mode* and (ii) a *step mode*. The interpretation mode applies a whole sequence of rules and tries to match the available rules as long as possible. In contrast to that, the step mode allows the transformation designer to select a certain rule or even a certain binding to be executed, i.e., the matching can be influenced by the transformation designer. As can be seen in Fig. 2.4, the transformation designer can use the *interactive match mode* to assign a certain element of a graph transformation rule to a certain element of the model graph. Only parts of the matches need to be specified whereas the remaining parts can be automatically completed. Although the user is allowed to choose an arbitrary rule in the step mode, it might happen that it cannot be bound in the current state of execution, thus breakpoints would nevertheless be needed, but are not supported in AGG. The current state of execution is shown in the according model graph. Furthermore, the

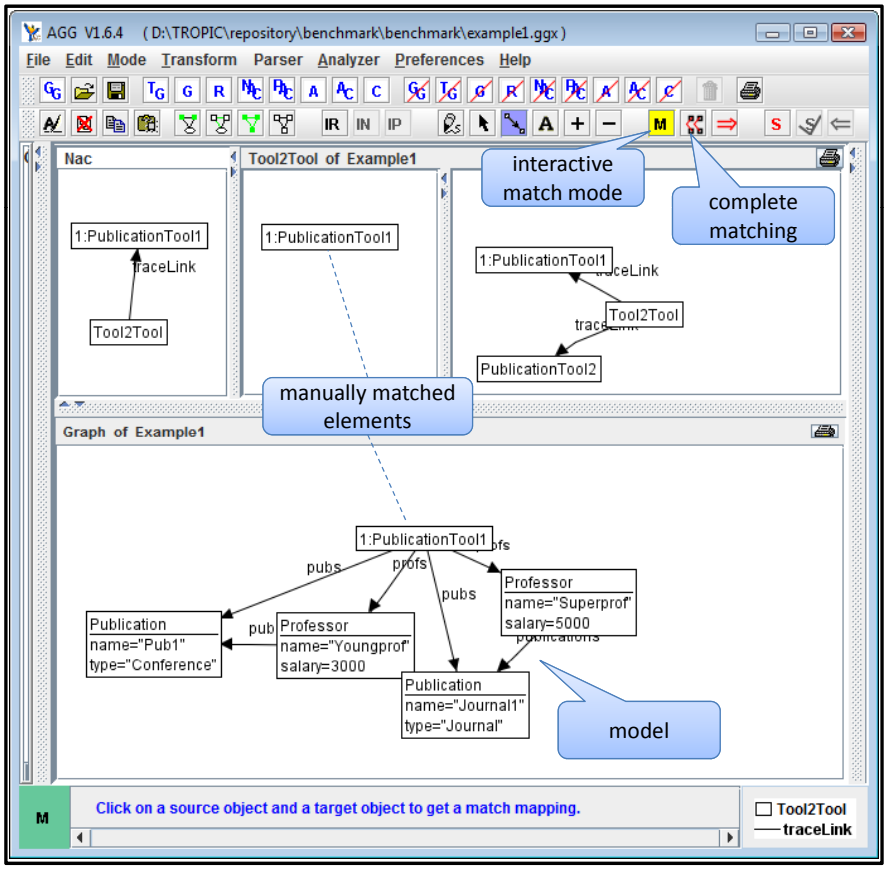


Figure 2.4: Screenshot of AGG Debugger.

model graph visualizes the current match. Adaptations to the model and the transformation logic are possible in AGG, but only if the according elements are not already matched by a certain rule. Finally, no support for forensic debugging is considered since AGG does not maintain a built in trace model, instead this has to be defined by the transformation designer.

Fujaba: Fujaba provides so-called story diagrams that allow to describe model transformations based on graph transformation rules. These graph transformation rules are then mapped to Java for execution. In this respect, the debugging features provided by a Java IDE may be reused. Nevertheless, in order to avoid the deficiencies of this approach, i.e., debugging on generated source code is difficult since the developer usually does not know about the generated code, in [49] an approach is presented that allows debugging on the story diagram level. Thereby, annotations are added to the generated Java code, which allows to reuse existing Java debuggers, but to map back the results to the model level. This mapping allows then to put breakpoints on story diagrams, whereby only a first proposal for conditional breakpoints exists [84]. Concerning investigation support, the variables values are shown to represent the according transformation state, together with highlighting the according elements in the story diagram. In [49] it is proposed to visualize the current heap of a Java program at runtime as a UML object diagram by means of eDOBS [50] to additionally reproduce a graphical representation of the model (cf.

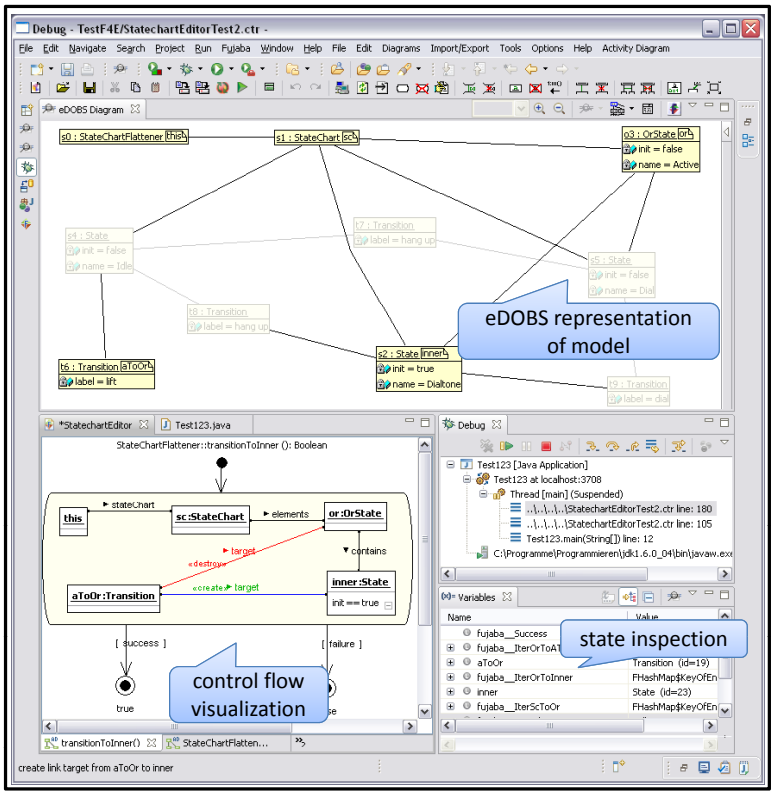


Figure 2.5: Screenshot of Fujaba Debugger with eDOBS [49]

Fig. 2.5). To execute the specified transformation stepwise, the underlying Java debugging features are used, which may lead to the fact that for a single step in the story diagram several steps are required in the underlying Java debugger. Finally, reasoning backwards is mentioned in [84] to be useful when debugging story diagram, but no actual realization is presented. Finally, first ideas are presented in which ways a model might be changed, i.e., only those elements might be changed that are not already matched by a certain story diagram.

GReAT: The graph transformation language GReAT provides a debugger which is built on top of the GReAT execution engine. Nevertheless, it only offers the typical features found in traditional debuggers, e.g., breakpoints and stepwise execution [6]. A graphical window displays a list of the transformation rules. The transformation designer can attach breakpoints to rules and step through the transformation specification, allowing to see the results of a particular rule or to see what elements are matched at a given time, as depicted in Fig. 2.6. During debugging, it is not possible to change model elements or the transformation specification. Finally, forensic debugging is not considered.

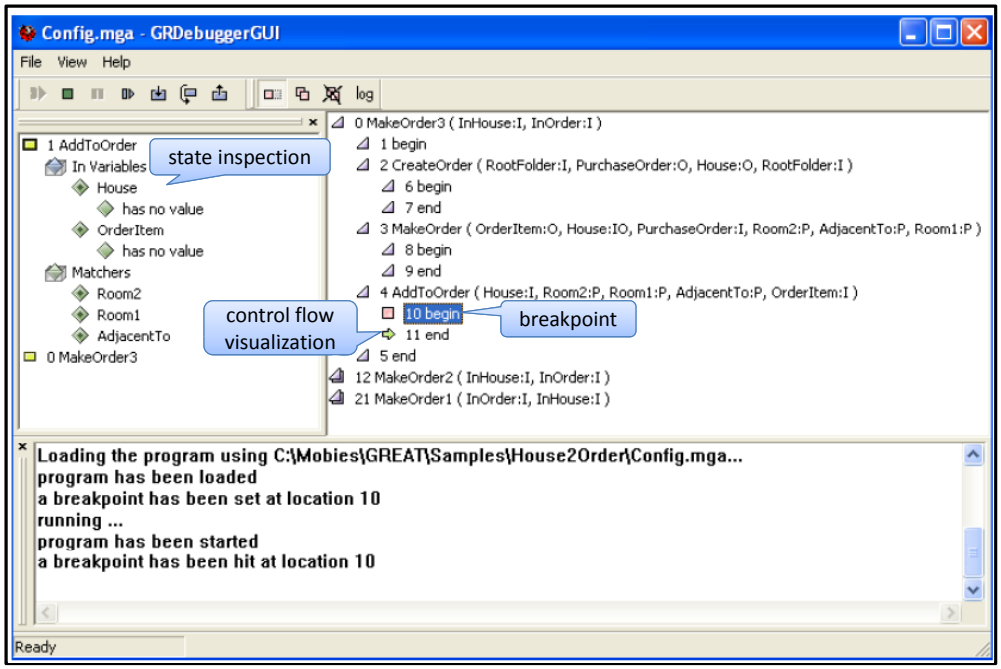


Figure 2.6: Screenshot of GReAT Debugger [6]

TGGs: Since TGG rules can be compiled to the Fujaba environment, using the MoTE plugin [160], TGGs can benefit of the debugging support of Fujaba as well, but only on the level of story diagrams and not on the level of TGGs. Therefore in [137] proposals have been made how debugging may be enabled on the level of TGGs by aligning debugging features of programming languages to TGGs. In Table 2.1, the according concepts are marked as proposed only. With the exception of AGG, the proposed debugging support is the only one that considers the matching of model elements as well. Furthermore, the proposal mentions that forensic debug-

ging could be employed in the context of TGGs since the traces are made explicit by means of the correspondence graph.

QVT Relations: For QVT Relations only the implementation of mediniQVT provides dedicated debugging support. Nevertheless, the debugger is based on the Eclipse debugging environment only. In this respect, no specific debugging support is provided to debug the matching process, i.e., it is not possible to select certain model elements or a certain rule. Nevertheless, besides simple breakpoints also conditional breakpoints are supported which allow the transformation designer to customize when the execution of a QVT Relations transformation should stop by means of OCL conditions. Inspection of the state is again limited to values of variables, only (cf. Fig. 2.7). In addition to highlighting according lines of code, the debug view shows the stack of called relations, i.e., if one relation calls another one, one can see its order of invocation. Furthermore, it is possible to use the known step semantics, i.e., *step into* steps into a dependent relation, *step over* solely executes the relation and returns the result and *step out* returns to the parent relation. Finally, QVT Relations allows to alter the variables during debugging, i.e., the model can be changed, but it is not possible to alter the transformation specification during debugging. Although the trace model, which is produced by the QVT engine is made explicit to the transformation designer, no support for forensic debugging is provided. In contrast to mediniQVT the second prominent QVT Relations tool ModelMorf [149] does not provide any explicit debugging facilities.

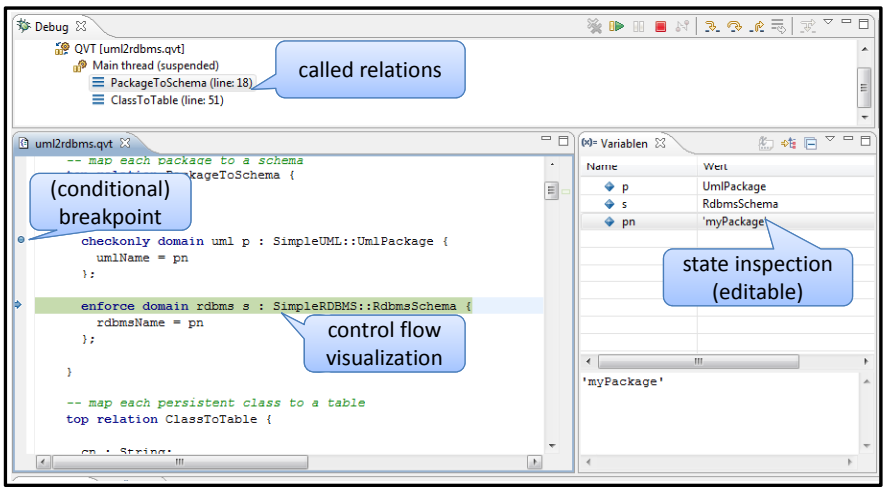


Figure 2.7: Screenshot of mediniQVT Debugger.

In summary, although numerous transformation languages claim debugging support for model transformations they typically make use of the debugging features of the underlying execution engine only. Thus, debugging occurs on a rather low-level and thus there is a considerable impedance mismatch between the high-level declarative specification of model transformations and the provided means for debugging. Especially, when debugging declarative languages this impedance mismatch hinders understandability, since the hidden operational semantics of declarative model transformation languages is not accordingly represented, e.g., only AGG pro-

vides support for debugging the matching phase of model transformations. Thus, what is needed is a debugger that represents all parts involved in a model transformation accordingly, i.e., not only the transformation specification itself but also the according metamodels and the models being transformed. Furthermore, a generally applicable debugging framework would be beneficial in order to provide a common debugger for various declarative model-to-model transformation languages.

2.4 Summary

In this section an overview on related work was provided. First a classification of existing approaches to model transformation testing was given. In the following, this classification was explained in detail and the various existing approaches have been discussed. A special focus was put on testing transformation by means of contracts, since this is a main contribution of the thesis, as will be discussed in Chapter 3. Nevertheless, testing might only help to observe the fact that there exists a failure in the transformation specification but does not necessarily provide means to detect the origin of the failure. This is why in a second step related work to runtime models and debugging has been considered. Since the envisioned runtime model serves as a basis for the proposed debugger, it was investigated if runtime models exist in the domain of model transformations. As no dedicated literature could be found in this domain, the scope was broadened to software engineering in general where runtime models have been considered to realize runtime verification. Finally, a classification of debugging facilities was proposed which was then used to investigate debugging support in existing model transformation languages. Based on the findings of the comparison to related work, the following chapter focuses on the first major contribution of this thesis being a visual, declarative language to specify contracts for model transformation testing.

Chapter 3

PaMoMo: A Visual Language for Model Transformation Contracts

Everything should be made as simple as possible,
but not simpler.

— *Albert Einstein*

Contents

In order to support the transformation designer to specify requirements of model transformations, this chapter introduces the visual, declarative language PaMoMo (Pattern-based Modeling Language for Model Transformations), which may be used to express model transformation contracts. In a first step, it is shown in which way contracts may be applied in the context of model transformations and the basic idea of PaMoMo is explained. After introducing the general idea, the language features are explained in detail and it is shown how reasoning on the specified contracts may be used to detect errors or inconsistencies within them, i.e., to check well-formedness of contracts. Since QVT Relations is used to operationalize the contracts, first the basic concepts of QVT Relations are introduced by providing an initial transformation specification of the running example before the actual translation of contracts specified in PaMoMo

to QVT Relations is discussed. Finally, this section concludes by showing how the contracts are executed and how the may thus be used to test a certain transformation specification.

3.1 Requirements Specification for Model Transformations

Requirements of model transformations should be made explicit, similar to software engineering in general, as emphasized in Section 1.3. In the area of software engineering, dedicated methods have been proposed to capture the requirements, ranging from informal methods, e.g., UML models like use-case diagrams, to more formal methods like Z [143], Alloy [70], or SysML[1]. Use-case diagrams are typically used in the very first step of requirements analysis by specifying when and under which conditions certain behavior occurs [20]. For this a high-level graphical syntax is used which typically does not allow to automatically derive properties of a system that could be used e.g., in the testing phase. In contrast, formal methods such as Z or Alloy make use of mathematical notations to describe properties of a system in a precise way. In this respect, it should be described what a system does instead of how it is achieved. As stated in [143] "formal specification may serve as single reference point for those who investigate the customer's needs and those who test the results". Consequently, this means that if requirements are specified in a formal way they could be used in the testing phase to test the programmed systems automatically against the requirements, i.e, they might be used as an *oracle function* in testing. SysML tries to combine the benefits of a high-level graphical syntax (i.e., ease of understandability) with the benefits of formal methods and thus supports the specification, analysis, design, verification and validation of systems but on a more abstract level. Instead of mathematical notations the (graphical) syntax bases on UML diagrams and thus is more common to the software engineer. Nevertheless, such languages are specific to the domain of software engineering in the way that they are not capable to deal with the complex structure of models and that they are not focused to model transformations making the specification of requirements of model transformations complex.

3.1.1 Design by Contracts for Model Transformations

The methods proposed for software engineering are too general, i.e., they tend to target the whole software system. Therefore, on a more fine-grained level, *design by contract* [105] was introduced as a means to increase quality in terms of correctness and robustness of the constructed software. Design by contract allows to formalize requirements (in terms of contracts) which may be used to test the software, i.e., contracts on method specify valid input parameters and report an error in case of invalid values. Another advantage of contracts is that they allow defining *what* a piece of software does but not *how* it is done. Different levels of contracts may be distinguished comprising *syntactic contracts* and *behavioral semantic contracts* [15]. The former enforce syntactically valid programs. In the context of model transformations, syntactic contracts are specified by the source and target metamodels since they describe the types of the manipulated data, implying that the source and target models must conform to these types [108]. In contrast, behavioral semantic contracts put further restrictions on the required input models,

[1] http://www.omgsysml.org/

the produced output models as well as their combinations [108]. In the first place, behavioral semantic contracts may be used to *precisely* specify the conditions (going beyond metamodel constraints) to be satisfied by input models such that the transformation is applicable, i.e., *preconditions*. Second, they may be used to express whether or not an output model should contain certain configurations of elements, i.e., *postconditions*. Finally, they may be used to specify *what* conditions need to be satisfied by any pair of input/output models of a correct transformation, i.e., *invariants* of the transformation (cf. Fig. 3.1).

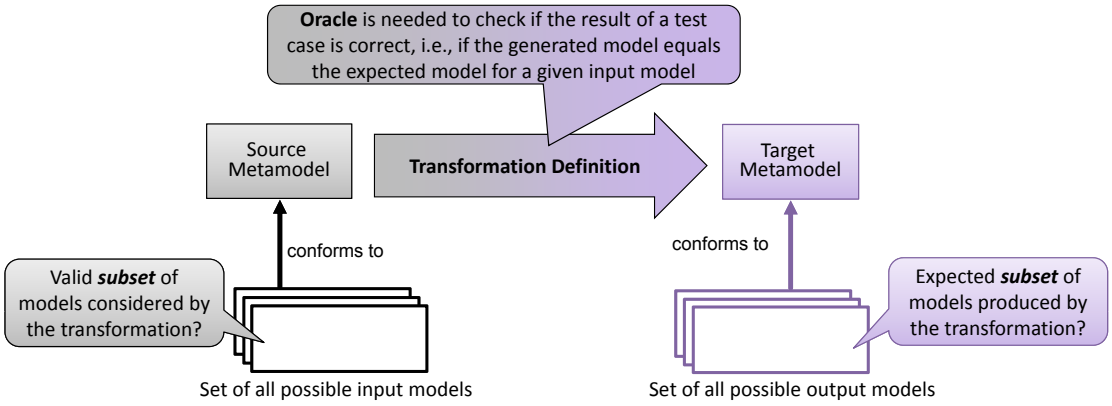


Figure 3.1: Contracts in Model Transformations

In the context of model transformations, contracts may be useful in several scenarios [30]:

- **Implementation:** A contract is a useful document for the transformation designer in the development phase, to make explicit the requirements that need to be implemented in a transformation.

- **Documentation:** Contracts serve as a useful documentation of the transformation in the maintenance phase. Moreover, if contracts have a formal semantics, they may be used to select transformations by matching properties of a required transformation and properties of transformations stored in a transformation library.

- **Compatibility Checking:** Contracts may be used to check the compatibility of transformations in a chaining scenario, e.g., to check whether the postconditions of a preceding transformation are compatible with the preconditions of a succeeding transformation.

- **Testing:** A common need in model transformation testing is to automatically compare expected output models to generated output models [98]. Unfortunately, the oracle that should predict the expected output models remains a major challenge [12], for which contracts (invariants) could be used to partially determine the expected output model (cf. Fig. 3.1).

3.1.2 Overview on PaMoMo

In order to make the requirements of model transformations explicit by the specification of contracts, in the following PaMoMo (Pattern-based Modeling Language for Model Transformations) is introduced. PaMoMo is a declarative, formal, visual language designed to express *behavioral semantic contracts* for transformations in an implementation-independent way [56]. The realization of contracts by a dedicated language has two main advantages though: (i) the definition of contracts is not tied to a particular target transformation language, i.e., is *implementation-independent*, which is especially favorable in MDE since no dedicated standard transformation language has been brought forward so far [38] and (ii) designers of transformations may make explicit desired properties of a transformation *before* implementation, which may be used for guiding the implementation. The contracts specified in PaMoMo may be beneficial in each of the above discussed scenarios. The focus of this thesis is on these parts of a contract comprising the testing scenario, i.e., how preconditions, invariants and postconditions may be applied to test model transformations.

Fig. 3.2 outlines the basic approach. First, the transformation designer uses PaMoMo to define a contract specifying preconditions, postconditions, and invariants for the transformation (cf. ① in Fig. 3.2). This contract exhibits a formal semantics and may be analyzed to discover redundancies and contradictions in contracts, and to measure coverage of the involved meta-

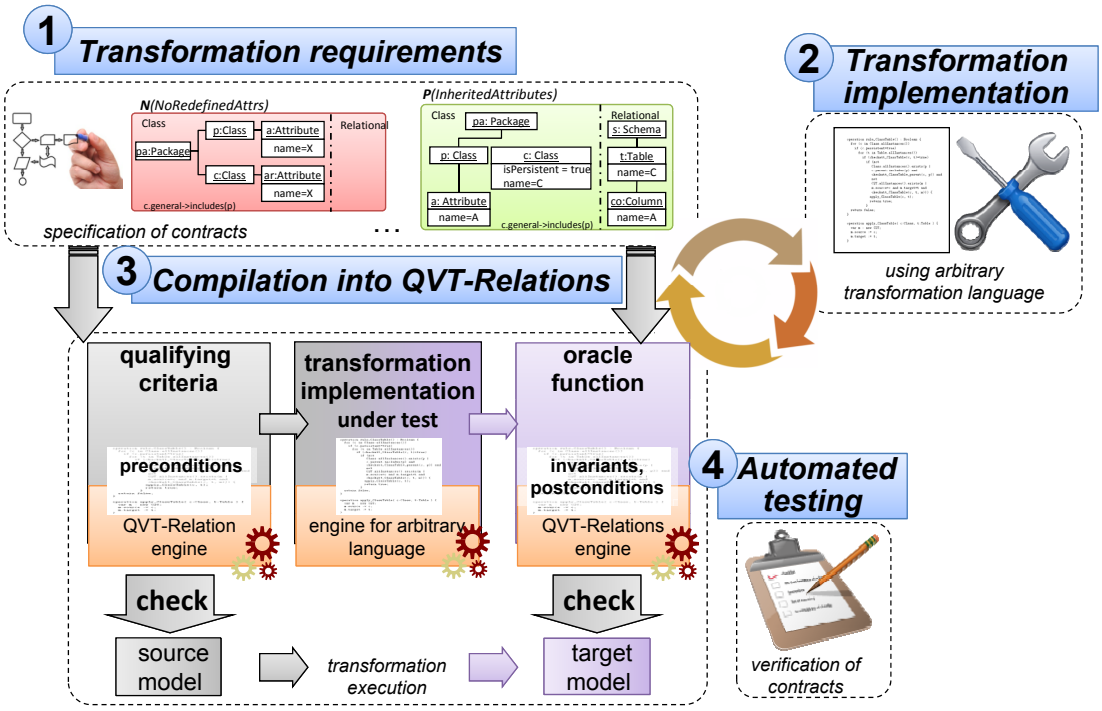


Figure 3.2: Automated Verification of Transformations using PaMoMo.

models, i.e., if every metamodel element is considered by a contract. Thus the well-formedness of several patterns and their correct interplay may be ensured. Next, the developer may use the contract as a high-level model to implement the transformation (cf. ② in Fig. 3.2). Although parts of the implementation may be (semi-)automatically derived from the contracts, this is not within the focus of this thesis. The implementation is tested by compiling the contract into the executable QVT Relations language (cf. ③ in Fig. 3.2), and then using a QVT engine in check-only mode in order to check if the transformed models fulfill the specified contracts. In this mode, a transformation is not used to produce a target model, but to check if a set of existing models conform to the transformation, and to report the locations where this is not the case. Hence, the compiled contract acts as an oracle describing invariants that output models should satisfy, and is used for automated testing (cf. ④ in Fig. 3.2). In this respect, first the validity of the input model is checked by executing the preconditions; next the transformation implementation may be executed; and finally it is checked if the input and resulting output models conform to the invariants and the postconditions. Consequently, the initial version of the PaMoMo language presented in [56] has been accordingly extended to provide a better language support for preconditions and postconditions (with enabling conditions), sets in invariants, and methods to reason at the pattern level. In the following, first the syntax and semantics of PaMoMo is provided, whereby the reader is referred to [56] for details on its formal semantics. Afterwards, the compilation into QVT Relations for the verification of a transformation implementation as a major new contribution is discussed.

3.2 Contract Specification with PaMoMo

After introducing the basic ideas, the syntax and semantics of PaMoMo is provided next. First, the modeling of *invariants* is explained and afterwards the modeling of *pre-* and *postconditions* is described, covering the requirements of the running example (cf. Fig. 1.2). Afterwards, *enabling* and *disabling conditions* of patterns are discussed, i.e., the satisfaction of a pattern is only demanded when certain conditions in the source *and* the target occur. Finally, sets are discussed, which allow to express properties related to the number of times a certain structure may occur in a model.

3.2.1 Modeling of Invariants

A PAMOMO contract consists of a set of declarative visual patterns. As stated before, PaMoMo allows to model preconditions concerning the source metamodel, invariants concerning the relationships between source and target metamodel and postconditions concerning the target metamodel. Therefore, patterns in PaMoMo are made of two compartments containing object graphs representing elements of the source or target metamodel. The left compartment contains objects typed on the source metamodel, e.g., `Class`, while the objects to the right are typed on the target metamodel, e.g., `Relational` (cf. Fig. 3.3). Consequently, patterns where only the left compartment is not empty are called *preconditions*, patterns where both the source and target compartments are not empty are called *invariants* and patterns where only the right compartment is not empty are called *postconditions*. To allow the transformation designer to describe

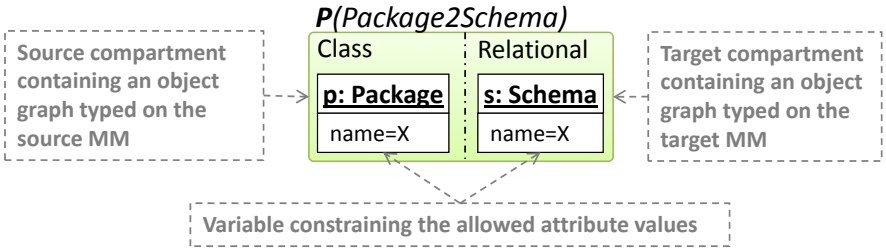


Figure 3.3: Positive Invariant Formalizing *Requirement 1*

necessary conditions to happen *positive* patterns are provided, i.e., the pattern is satisfied by a pair of models if these contain certain elements. To additionally allow to express forbidden situations additionally *negative* patterns are provided, i.e., the pattern is satisfied if certain elements are not found in the models. As an example, Fig. 3.3 shows a positive pattern formalizing requirement 1 of the example transformation. Positive patterns are represented in green with its name enclosed in `P(...)`, while negative patterns are shown in red with its name enclosed in `N(...)`. In order to consider attributes values in contracts, objects in the source and target compartments may have attributes that may be assigned either a concrete value, or a variable (like X in the example). A variable may be assigned to several attributes to ensure equality of their values, or may be used in the pattern constraint expression. These may involve elements of the source and target compartments. The invariant of Fig. 3.3 has no expression, but variable X is assigned to the name of the package and the schema, hence requiring the equality of both names. Nevertheless, to allow for more complex comparisons, a constraint expression using the Object Constraint Language (OCL) [115] may be specified.

Fig. 3.4 shows a scheme of the satisfaction of a positive (cf. Fig. 3.4(a)) and a negative invariant (cf. Fig. 3.4(b)) over a pair of models, where `EXP` represents the pattern constraint expression. Thus, the satisfaction for positive invariants amounts to check:

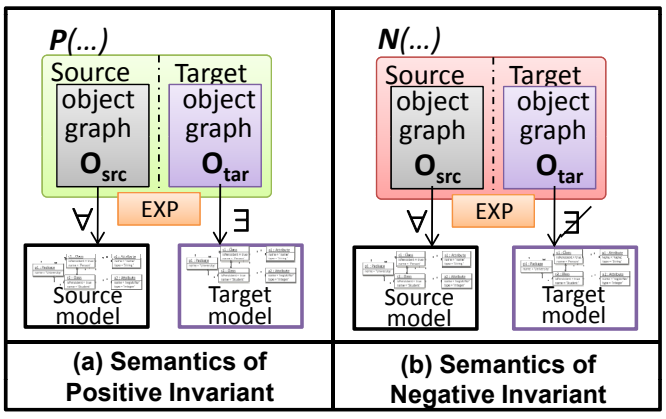


Figure 3.4: Scheme of the Semantics of Positive and Negative Invariants

$$\forall Occ(O_{src})\ s.t.\ EXP|_{src}(Occ(O_{src}))$$
$$\exists Occ(O_{tar})\ s.t.\ EXP(Occ(O_{src}), Occ(O_{tar}))$$

where $EXP|_{src}$ is the part of the expression `EXP` that contains source objects, attributes and variables only, and $Occ(O_{src})$, $Occ(O_{tar})$ represent an occurrence of the source and target object graphs respectively. An occurrence is a binding from the objects in the object graph of the pattern to elements in the model. A pattern invariant is therefore satisfied either if no occurrence of the source object graph of the pattern may be found (called *vacuous satisfaction*) or if for each occurrence of the source object graph, a corresponding occurrence of the target object graph is found (or not found if the invariant is negative). A contract is satisfied if *all* its patterns are satisfied, hence a *conjunction* is assumed between all the patterns of the contract. The application of the pattern covering requirement 1 presented in Fig. 3.3 is shown in Fig. 3.5(a). The first example models fulfill the specified requirements since for the package `p1` in the source model there is an equally named schema `s1` in the target models. The second scenario is true since if no package is found in the source model then it is impossible to check if there is an according schema in the target model. Thus, in this case the invariant vacuously holds. Finally, the third scenario fails since the only existing university object is differently named than the source package. If the positive invariant is changed to a negative one, then scenario one fails but scenario three succeeds since in this case no equally named packages and schemas may exist (cf. Fig. 3.5(b)).

Fig. 3.6 shows the invariants addressing requirements 2, 3 and 4 of the running example (i.e., transformation of classes, attributes and inherited attributes). The invariant to the left states that for each persistent class `c` in a package `p`, there must be an equally named table `t` in a corresponding schema `s`. The invariant in the middle states that each attribute `a` of a persistent class must be transformed into a column `co` with the same name and type. Finally, the right-most invariant states that if a class `c` has a superclass `p` owning an attribute `a`, then the table `t` that corresponds to `c` must contain a column with the same name as the attribute. This invariant con-

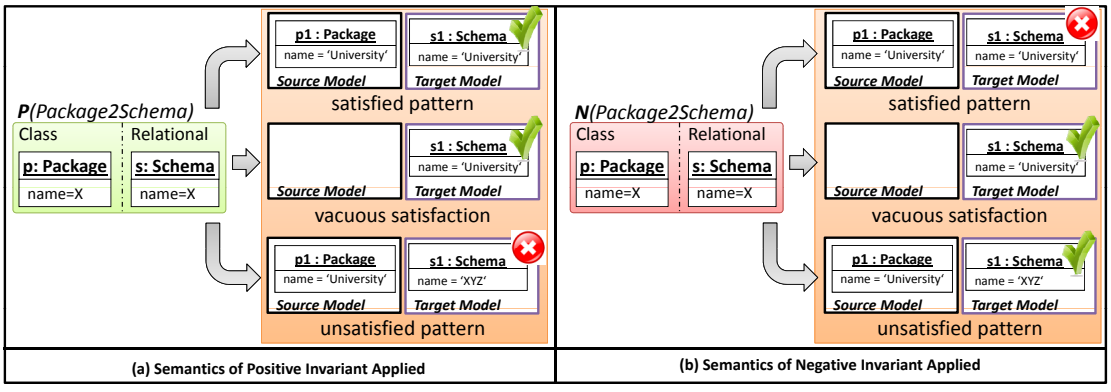


Figure 3.5: Semantics of Positive and Negative Invariants Applied

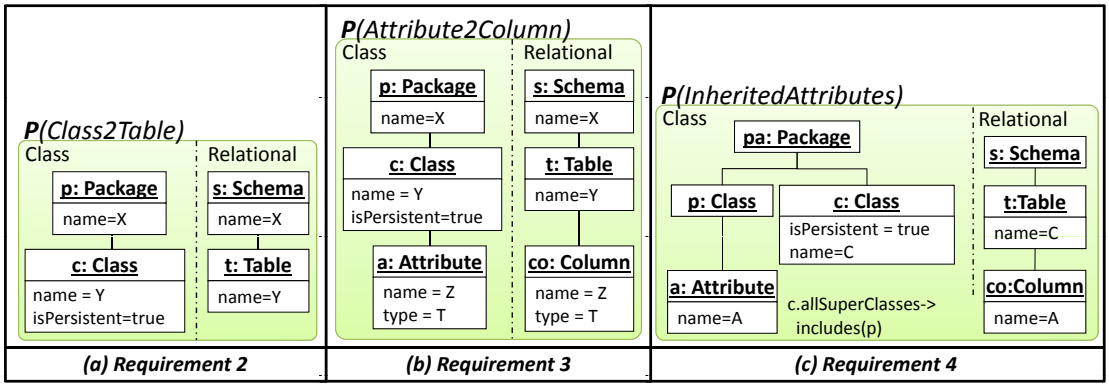


Figure 3.6: Additional Invariants Formalizing *Requirements 2*, *3* and *4*

tains a constraint checking that the derived property `allSuperClasses` of class `c` includes the class `p` (i.e., `p` is a superclass of `c`).

3.2.2 Modeling of Preconditions and Postconditions

In contrast to invariants, which relate source and target models, (i.e., both compartments contain object graphs), preconditions refer only to elements of the source metamodel (i.e., only the source compartment of the pattern contains an object graph) and postconditions refer only to elements of the target metamodel (i.e., only the target compartment contains an object graph). The left side of Fig. 3.7 shows a precondition expressing requirement 5 in the example (i.e., absence of redefined attributes in class hierarchies) by a negative pattern. The right part of the figure shows the postcondition to express requirement 6 (i.e., absence of duplicated columns in the same table) as a negative pattern as well.

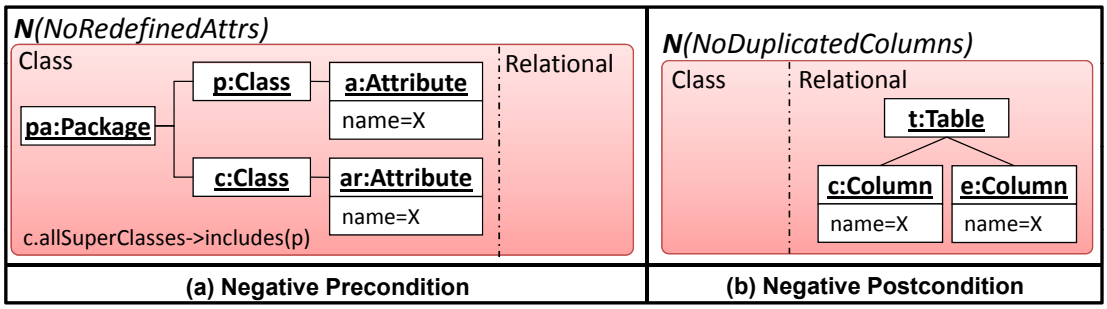


Figure 3.7: Precondition (*Requirement 5*) and Postcondition (*Requirement 6*)

Fig. 3.8 depicts the semantics of positive and negative preconditions. Positive preconditions demand the existence of a structure in the source model satisfying the expression constraint. Negative preconditions demand the absence of a structure in the source model satisfying the expression constraint. Postconditions have equal semantics, but are evaluated on the target model.

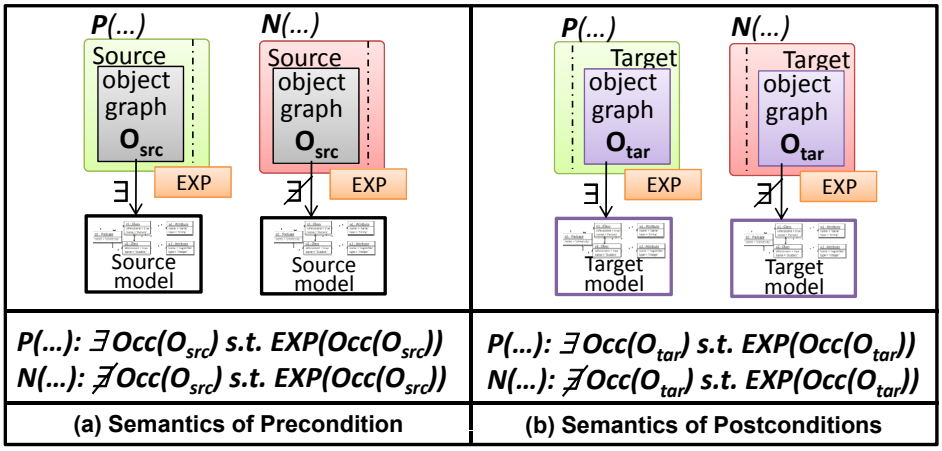


Figure 3.8: Scheme of the Semantics of Preconditions and Postconditions

To explicate the semantics of preconditions, Fig. 3.9 shows the pattern modeling requirement 5 of our running example applied to several source models. Thereby, the pattern holds in the first scenario since class `c1` is a superclass of `c2` and their according attributes are differently named. In the second scenario the pattern also holds (but only vacuously) as the pattern may not be applied as there is no package available in the source model. In the third scenario the precondition fails since the attribute `a1` and `a2` exhibit equal names and their according classes (`c1` and `c2`) inherit from each other.

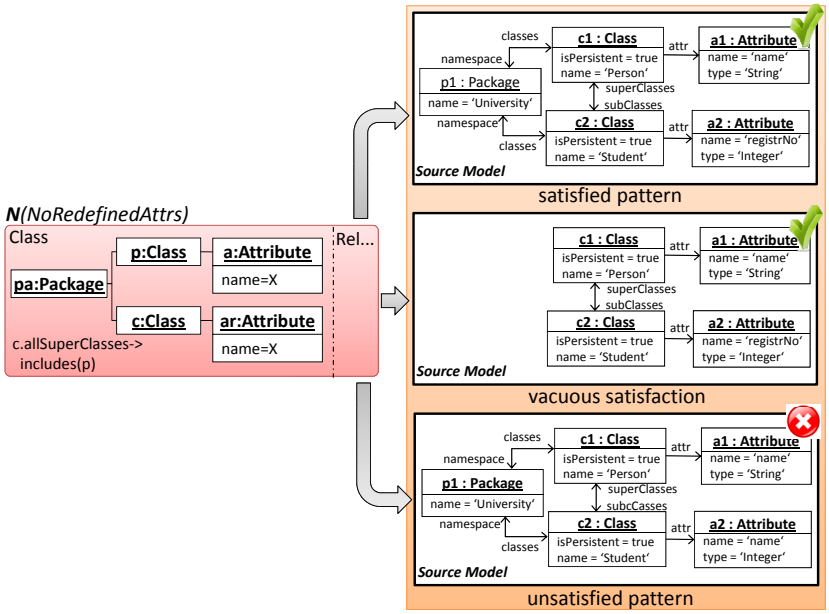


Figure 3.9: Semantics of Negative Precondition Applied

In order to also explicate the semantics of postconditions, Fig. 3.10 depicts a simple example which requires that tables have to have a name. In this respect, again the first postcondition is fulfilled since both tables exhibit a name whereas the second scenario holds only vacuously since no table is available in the target model. Finally, the postconditions fails in the third scenario since table `t2` has no name.

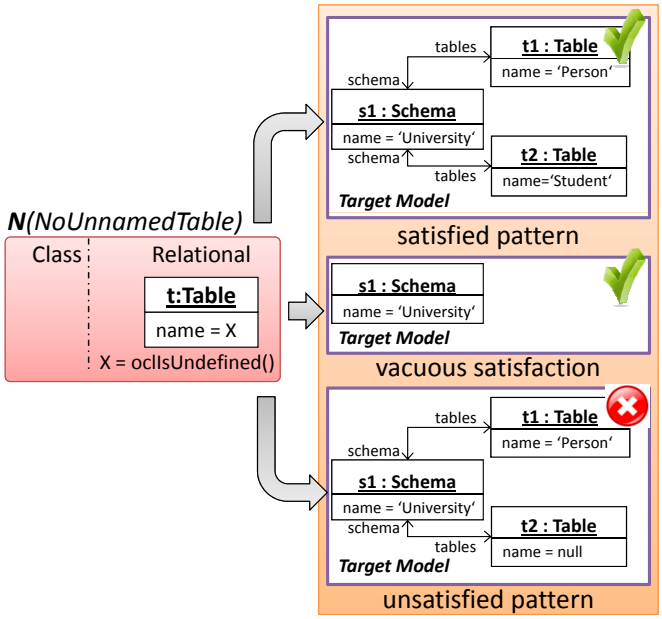


Figure 3.10: Semantics of Negative Postcondition Applied

3.2.3 Modeling of Enabling and Disabling Conditions for Patterns

The patterns presented so far checked that for all occurrences of an object graph in the source model, a corresponding structure in the target exists. However, some more flexibility is often needed to demand the satisfaction of a pattern only when certain conditions in the source *and* the target occur, e.g., only if a package contains at least one persistent class, then a schema has to be created. For this purpose, patterns may define *enabling* and *disabling* conditions, which restrict their satisfaction.

In particular, enabling and disabling conditions allow expressing properties which need to hold only if the premise of an implication is fulfilled. Each pattern may define any number of disabling conditions and one enabling condition, i.e., premisses. This permits formulating properties of the form $if\ \langle enabling \rangle\ and\ (not\ \langle disabling_1 \rangle)\ ...\ and\ (not\ \langle disabling_n \rangle)\ then\ \langle pattern \rangle$. For instance, Fig. 3.11 shows an invariant with an enabling condition to the left, so that the invariant is required to be satisfied only for packages for which there is an equally named schema. In such a case, the invariant states that the transient classes inside the packages should not have a corresponding table in the schema (because the invariant is negative). This pattern

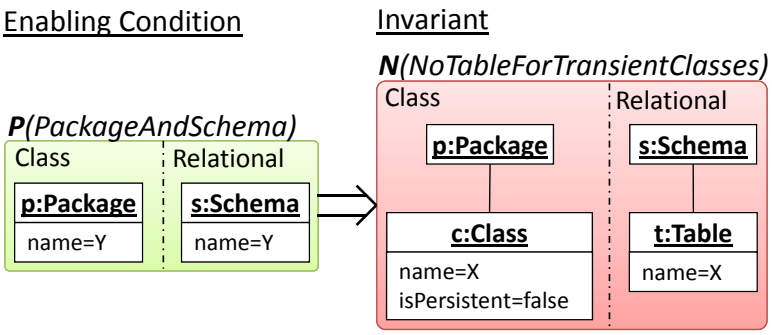


Figure 3.11: Invariant with Enabling Condition

uses a non-constructive specification style, ensuring that a transformation implementation will not accidentally translate a non-persistent class into a table.

Fig. 3.12 shows to the left the scheme of an invariant with one enabling and one disabling condition, while the right part sketches its evaluation on a pair of models. The pattern first looks for all occurrences of the source object graph of the invariant *plus* the enabling condition, which additionally (i) fulfill the expression EXP^{EN} of the enabling condition, and (ii) fulfill the part of the invariant expression containing only source elements ($EXP|_{src}$), and (iii) for which no occurrence of the disabling condition (which might contain an expression EXP^{DS}) is found. Then, for each one of these occurrences, i.e., *for all* semantics, there should be an occurrence of the target object graph of the invariant satisfying the invariant expression. Please note that enabling and disabling conditions permit including target elements in the pattern condition.

The evaluation of invariants with enabling and disabling conditions is therefore pursued as follows:

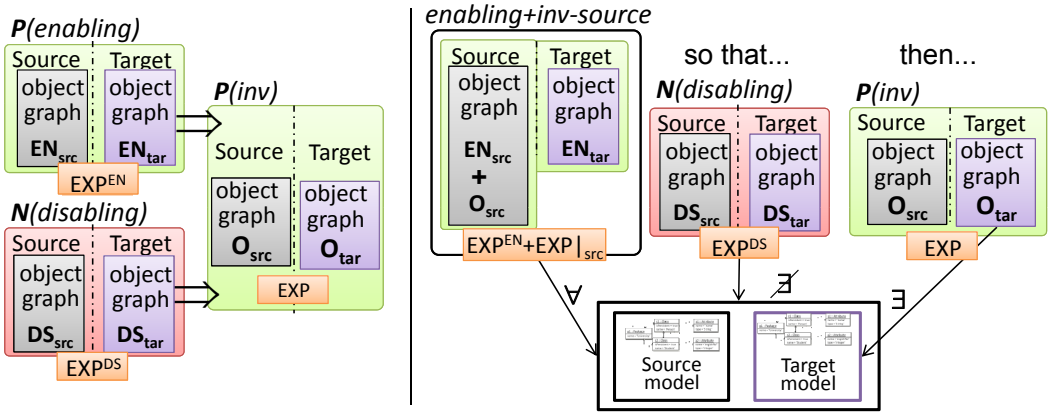


Figure 3.12: Scheme of the Semantics of Enabling and Disabling Conditions

$$\forall Occ(EN_{src} + O_{src}, EN_{tar})\ s.t.$$
$$[(EXP^{EN} + EXP|_{src})(Occ(EN_{src} + O_{src}, EN_{tar})) \wedge$$
$$\nexists Occ(DS_{src}, DS_{tar})\ s.t.\ EXP^{DS}(Occ(DS_{src}, DS_{tar})) \wedge ...]$$
$$\exists Occ(O_{tar})\ s.t.\ EXP(Occ(O_{src}), Occ(O_{tar}))$$

Fig. 3.13 illustrates how enabling conditions modify the semantics of a pattern, through an example of two syntactically similar invariants for classes, one declaring an enabling condition and the other not. The invariant in the lower left demands the existence of a schema and table for each persistent class in a package. The models shown above fulfill this, as the class model contains two occurrences of the source of the invariant (i.e., two classes), and for each one a schema in the relational model may be found defining a table with same name as the class. In contrast, the models do not satisfy the invariant to the right. This is so as this invariant demands that for every occurrence of a persistent class, its package and equally named schema (this latter required by the enabling condition), a table with same name as the class exists. This is not true in this case as, for instance, for the objects `p`, `c1` and `s2`, there is no table named "Person" in `s2`.

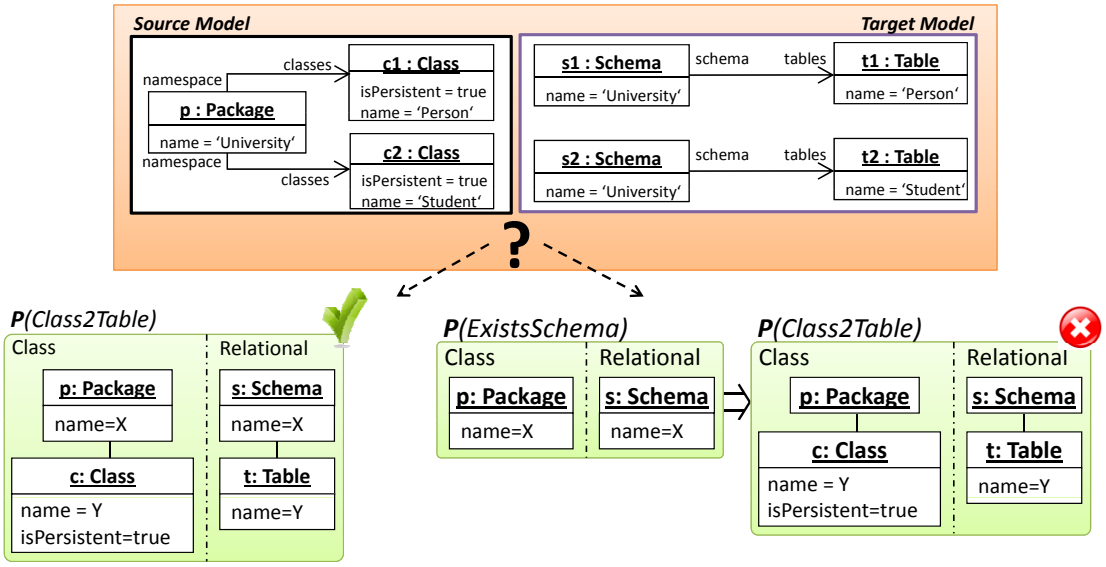


Figure 3.13: Semantics of Invariants with and without Enabling Condition

Pre- and postconditions may have enabling and disabling conditions as well. As an example, Fig. 3.14 shows the scheme of the semantic interpretation of a precondition with an enabling condition to the left. In this case, for each occurrence of the enabling condition, an occurrence of the precondition needs to be found. For the sake of illustration, the right part of the figure shows an example precondition demanding each persistent class to have at least one attribute.

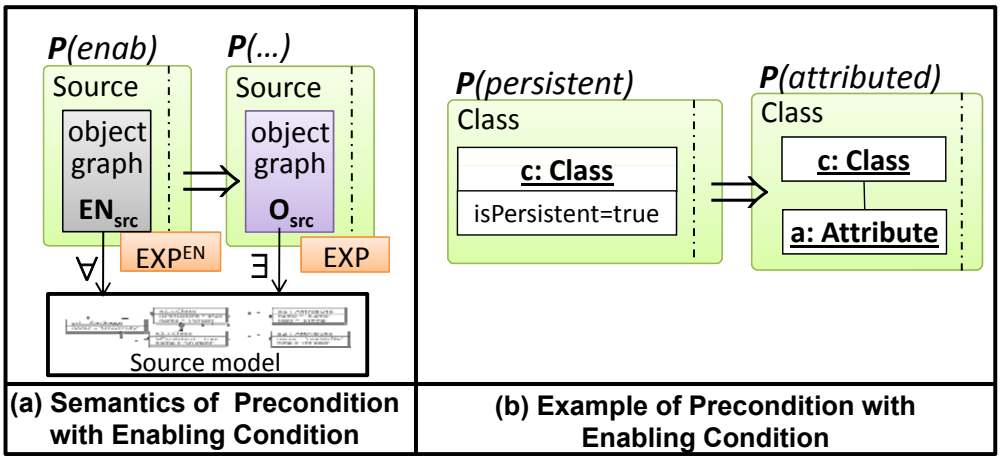


Figure 3.14: Precondition with Enabling Condition

3.2.4 Modeling Patterns for Collections of Model Elements

It is sometimes useful to formulate properties related to the number of times a certain structure may occur in a model. For this purpose, patterns may define variable sets of source and target elements (improving the expressive power compared to [56]). A set is depicted as a polygon with a name (see for example set `pclasses` in Fig. 3.15) and it represents the set of all occurrences of the structure enclosed in the polygon. Furthermore, sets may be nested and contain arbitrary structures.

As an example, the left side of Fig. 3.15 shows an invariant making use of sets in the source and target. The invariant states that the number of persistent classes in a package (size of set `pclasses`) should be the same as the number of tables in the corresponding schema (size of set `tables`). The center and right sides of Fig. 3.15 show the evaluation scheme of invariants

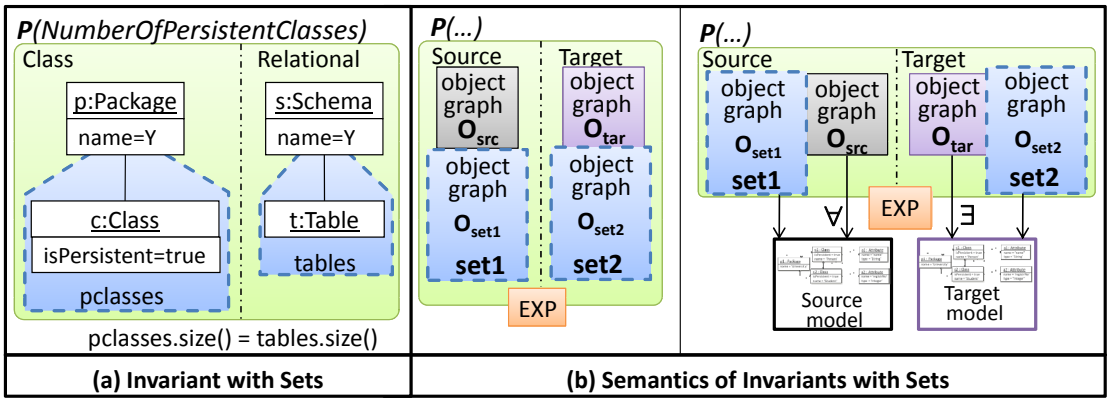


Figure 3.15: Invariant with Sets

with sets. The figure in the middle represents an invariant with two sets (`set1` in the source and `set2` in the target) and a constraint expression EXP that includes both sets. A pair of models satisfies such an invariant if for each occurrence of the source object graph, there is an occurrence of the target object graph that satisfies the constraint expression. Such an expression may make use of the sets `set1` and `set2` of all occurrences of the object graphs O_{set1} and O_{set2}:

$$\begin{aligned}&\forall Occ(O_{src})\ s.t.\ EXP|_{src}(O_{src}, Set\ of\ all\ Occ(O_{set1}))\\ &\quad \exists Occ(O_{tar})\ s.t.\\ &\qquad EXP(Occ(O_{src}), Occ(O_{tar}), Set\ of\ all\ Occ(O_{set1}), Set\ of\ all\ Occ(O_{set2}))\end{aligned}$$

3.2.5 PaMoMo Metamodel

The above presented concepts of PaMoMo have been specified on the basis of a metamodel describing the abstract syntax, which is depicted in Fig. 3.16. The class `Specification` represents the root container for all `Patterns` and allows to specify URIs to the source and target metamodels. The abstract base class `Pattern` stores the constraint expression (cf. attribute `Pattern.OCLexpression`). The two concrete subclasses `PositivePattern` and `NegativePattern` are used to distinguish between these two types of patterns. Fur-

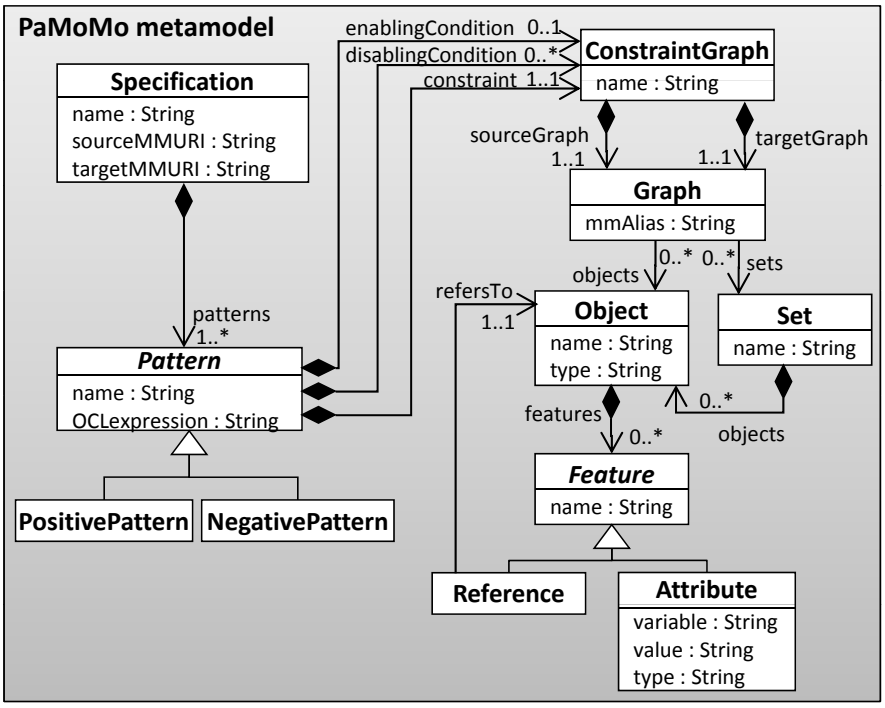


Figure 3.16: Metamodel of PaMoMo

thermore, the class `Pattern` refers to `ConstraintGraph` by means of three different roles: (i) constraint, (ii) enabling condition and (iii) disabling conditions which are used to model the according types of patterns as the names already imply. `ConstraintGraphs` act as a container for source and target `Graphs` (cf. references `ConstraintGraph.sourceGraph` and `ConstraintGraph.targetGraph`) which represent elements of the source or target metamodel. In this respect, a graph contains `Objects`, which might again contain `Features` (either `Reference` or `Attribute`) which merely represent aliases to the elements of the metamodel. Last but not least, a `Graph` might also contain `Sets` to model the set semantics of PaMoMo.

After discussing how to model patterns by using PaMoMo and representing their semantics, it is elaborated on reasoning techniques which allow to check well-formedness of several patterns and their interplay.

3.3 Reasoning with Patterns

In PAMOMO, contracts might exist of several patterns. Consequently, it has to be ensured that they are well-formed concerning their interplay, i.e., the contract may never be fulfilled if there exists a pattern with a negative pre- or postcondition and if this condition is included in a positive pre- or postcondition, then either one of the contract fails. In order to statically prevent such errors, the formal semantics of PAMOMO allows for reasoning on: (i) *metamodel coverage*, (ii) *redundancies*, (iii) *contradictions* and (iv) *pattern satisfaction* on contracts, as detailed in the following.

First, *metamodel coverage* means the identification of elements in the source and target metamodels that are used in a PAMOMO contract, as well as how they are used (i.e., in enabling or disabling conditions only, or in positive/negative patterns). This helps the transformation designer assess which parts of a source or target metamodel are referenced by a given set of patterns and allows for a quick identification of underspecifications, i.e., a transformation might transform elements that are not checked by any of the patterns and consequently their correctness may not be ensured. If, for instance, some element in the target metamodel is not used in any positive invariant, it is never checked if such an element gets created correctly. In the presented example in Figs. 3.3–3.15 all elements in both metamodels are used, i.e., full metamodel coverage is achieved.

Second, *redundancies* in contracts may be statically detected (cf. Table 3.1). A redundant pattern may be safely removed yielding a simpler, more compact contract with the same semantics. For instance, if a positive pre- or postcondition is included in a more comprehensive, positive pre- or postcondition, the smaller one is redundant and may be removed. The reason is that whenever the more comprehensive one is found, the smaller one will be found as well. Similarly, if a negative pre- or postcondition is included in a more comprehensive one, the more comprehensive one is redundant. Table 3.1 shows these two redundancy cases (first row), as well as redundancies that may be identified for invariants (second row) and for the disabling conditions of a pattern (third row). For example, if a pattern has a disabling condition included in another one, then the more comprehensive condition is redundant.

Table 3.1: Detection of redundancies in PAMOMO contracts. P_i and N_i are a positive and a negative pattern without enabling or disabling conditions.

Scope	Redundancy
Pre/Postconditions	$P_1 \subseteq P_2 \Rightarrow P_1$ is redundant
	$N_1 \subseteq N_2 \Rightarrow N_2$ is redundant
Invariants	$P_{1,src} = P_{2,src}$ and $P_{1,tar} \subseteq P_{2,tar} \Rightarrow P_1$ is redundant
	$P_{1,tar} = P_{2,tar}$ and $P_{1,src} \subseteq P_{2,src} \Rightarrow P_2$ is redundant
	$N_{1,src} = N_{2,src}$ and $N_{1,tar} \subseteq N_{2,tar} \Rightarrow N_2$ is redundant
	$N_{1,tar} = N_{2,tar}$ and $N_{1,src} \subseteq N_{2,src} \Rightarrow N_2$ is redundant
Enabling/Disabling Conditions of a Pattern	$disabling_1 \subseteq disabling_2 \Rightarrow disabling_2$ is redundant

Third, *contradictions* which prevent the satisfaction of a contract by any pair of models (cf. Table 3.2) may be statically detected. For example, there is a contradiction if a negative pre- or postcondition is included in a positive pre- or postcondition. The reason is that the satisfaction of the positive precondition requires finding an occurrence in the source model, but this means that an occurrence of the negative precondition will be found as well. This conflict corresponds to the first row in Table 3.2. The table includes another contradiction that may arise when two invariants have the same source, one is positive and the other is negative, and the target of the negative one is included in the target of the positive one. In this case the invariants may not be simultaneously satisfied whenever an occurrence of their source part is found in the source model. Only if a source model does not contain the source part of the invariants these would hold (vacuous satisfaction).

Table 3.2: Detection of contradictions in PAMOMO contracts. P and N are a positive and a negative pattern without enabling or disabling conditions. Subindex src and tar refer to the source and target of a pattern.

Scope	Contradiction
Pre/Postconditions	$N \subseteq P \Rightarrow$ contract is unsatisfiable
Invariants	$N_{src} = P_{src}$ and $N_{tar} \subseteq P_{tar} \Rightarrow$ contract is potentially unsatisfiable

Finally, reasoning on the *satisfaction* of patterns is enabled in order to detect potential errors in a contract and reporting a warning. For instance, consider a negative precondition that is included in the source part of an invariant or in one of its enabling conditions. In this case there is no contradiction, but if the negative precondition holds, then the invariant will also hold vacuously because it will never be enabled. If the precondition does not hold, then the invariant may be satisfied or not (depending on whether its main pattern is found in the models). Nevertheless the whole contract will not hold. Thus, this situation usually indicates an error in the specification. Table 3.3 gathers different warnings for PAMOMO contracts concerning satisfiability.

Table 3.3: Detection of potential errors in PAMOMO contracts concerning satisfiability. $NPre$ and $NPos$ are a negative precondition and a negative postcondition. I is a (positive or negative) invariant. Subindex src and tar refer to the source and target of an invariant.

Scope	Satisfaction
Pre/Postconditions	$N_{Pre} \subseteq I_{src} \Rightarrow$ if N_{Pre} holds, I vacuously holds
	$N_{Pre} \subseteq I_{enabling} \Rightarrow$ if N_{Pre} holds I vacuously holds
	$N_{Pos} \subseteq I_{tar} \Rightarrow$ if N_{Pos} holds I vacuously holds
	$N_{Pos} \subseteq I_{enabling} \Rightarrow$ if N_{Pos} holds I vacuously holds
Enabling/Disabling Conditions of a Pattern	*disabling* $\subseteq$ *enabling* $\Rightarrow$ pattern vacuously holds

As an example of this kind of reasoning, Fig. 3.17 shows a negative precondition on top discarding the transformation of models where some package contains duplicated classes. The invariant below, deals with the transformation of equally named classes inside a package, which should be transformed into a single table containing columns for the attributes of the classes. Thus, the second invariant is useless because it may only be satisfied (in a non-vacuous way) if the input model has duplicated classes, but this is forbidden by the negative precondition. This situation, which corresponds to the first row in Table 3.3 (i.e., $NPre \subseteq I_{src}$), gives rise to a warning.

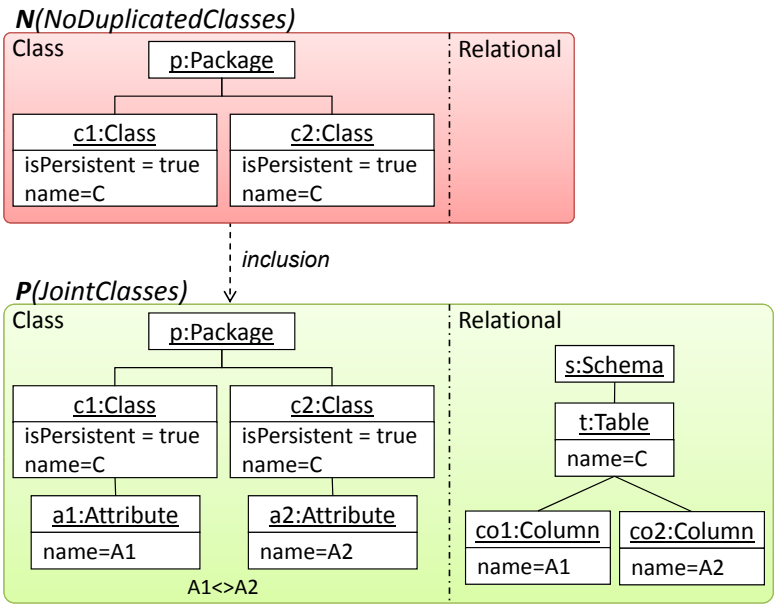


Figure 3.17: Potential Error: Disabled Invariant due to Negative Precondition

3.4 QVT Relations in a Nutshell

After the designer has specified the transformation requirements in terms of contracts (cf. step 1 in Fig. 3.2), the developer may start implementing the model transformation (cf. step 2 in Fig. 3.2). Although any arbitrary transformation language might be chosen for this task, QVT Relations is employed in our running example. This is since QVT Relations is also used to automatically verify the specified contracts (cf. Section 3.5), and thus, the reader is not confronted with many different languages.

QVT Relations is a declarative model transformation language standardized by the Object Management Group (OMG) [116]. It allows for several execution scenarios, like *model transformation* (i.e., generating a new target model from an existing source model), *model synchronization* (i.e., synchronizing two existing models) and *consistency checking* (i.e., checking the synchronization of two existing models without enforcing it).

With QVT Relations, a transformation is specified as a set of relations that must hold between a set of models, called *candidate models* in the QVT standard. Each relation defines local constraints to be satisfied by the candidate models, and has two or more domains. Domains are described by object graph patterns, and have a flag to indicate whether they are `checkonly` or `enforce`. The models of a domain marked as `enforce` may be modified in order to satisfy the relation. In contrast, the models of a domain marked as `checkonly` are just inspected to check if the relation holds for the candidate models, resulting in reported errors only. Thus, in order to realize a transformation scenario, the target domain must be marked as `enforce` to allow the creation of a new target model, and the transformation must be executed in the direction of this domain. In the example transformation, the aim is to generate a new target model from an existing source model, and hence the domain `class` is marked as `checkonly` whereas the domain `rel` is marked as `enforce`.

Fig. 3.18 shows a first version of the QVT Relations implementation for the running example. The transformation comprises two candidate models `class` and `rel` (cf. line 2) representing a model conforming to the `Class` metamodel and a model conforming to the `Relational` metamodel, respectively. The transformation specification contains five relations, namely `PackageToSchema`, `ClassToTable`, `AttributeToColumn`, `PrimitiveAttributeToColumn` and `SuperAttributeToColumn`. Relations may be *top-level* or not, which is indicated with the keyword `top`. The execution of a transformation requires that all its top-level relations hold. In contrast to this the non-top level ones only need to hold when they are invoked directly or indirectly from top-level relations. A relation holds if for each binding of the objects in the source graph pattern (in the source model), there exists a valid binding of the target pattern objects (in the target model).

Assuming that the execution starts with the top relation `ClassToTable` in the example (cf. line 16), it is required that for each persistent class `c` contained in a package `p`, a table `t` contained in a schema `s` exists. Furthermore, the class `c` and the table `t` must be equally named, which is enforced by using a common variable `cn`.

In addition, relations may contain *when* and *where* clauses. The former express preconditions under which the relation needs to hold. They usually refer to other relations, to which they pass a number of parameters that appear as variables in the current relation. For instance,

```
1 transformation ClassToRel
2  (class : Class ; rel : Relational ){
3
4  // map each package to a schema
5  top relation PackageToSchema {
6   pn: String ;
7   checkonly domain class p: Package {
8    name =pn
9   };
10   enforce domain rel s: Schema {
11   name =pn
12  };
13 }
14
15 // map each persistent class to a table
16 top relation ClassToTable {
17  cn: String ;
18  checkonly domain class c: Class {
19   namespace =p: Package {},
20   isPersistent =true ,
21   name =cn
22  };
23  enforce domain rel t: Table {
24   schema =s: Schema {},
25   name =cn
26  };
27  when {
28   PackageToSchema (p, s);
29  }
30  where {
31   AttributeToColumn (c, t);
32  }
33 }
34 // map each attribute to a column
35 relation AttributeToColumn {
36  checkonly domain class c: Class {};
37  enforce domain rel t: Table {};
38  where {
39   PrimitiveAttributeToColumn (c, t);
40   SuperAttributeToColumn (c, t);
41  }
42 }
43
44 // map each attribute to a column
45 relation PrimitiveAttributeToColumn {
46  an , tn: String ;
47  checkonly domain class c: Class {
48   attributes =a: Attribute {
49    name =an,
50    type =tn
51   }
52  };
53  enforce domain rel t: Table {
54   columns =cl: Column {
55    name =an ,
56    type =tn
57   }
58  };
59 }
60
61 // map inherited attributes
62 relation SuperAttributeToColumn {
63  checkonly domain class c: Class {
64   superclasses=sc: Class {}
65  };
66  enforce domain rel t: Table {};
67  where {
68   SuperAttributeToColumn (sc , t);
69  }
70 }
71 }
```

Figure 3.18: Class2Relational Transformation Implemented in QVT Relations

the relation `ClassToTable` is only required to hold if the relation `PackageToSchema` holds, as this latter relation appears in the *when* clause of the `ClassToTable` relation (cf. line 28). *Where* clauses are used to specify relation postconditions, i.e., if the current relation holds then the *where* clause should hold, and may also include references to other relations. For instance, `ClassToTable` requires the relation `AttributeToColumn` to hold in its *where* clause (cf. line 31). This second relation delegates the transformation of attributes to the relations `PrimitiveAttributeToColumn` and `SuperAttributeToColumn` in its *where* clause (cf. lines 39 and 40). The relation `PrimitiveAttributeToColumn` transforms the attributes of a class `c` into equally named and typed columns of the corresponding table. Finally, the relation `SuperAttributeToColumn` deals with inherited attributes by recursively calling itself (cf. line 68), i.e., attributes of super classes should be transformed as well and should be added to the according table stemming from the subclass.

As the attentive reader might have already spotted, by the recursive call in the *where* clause of the `SuperAttributeToColumn` relation, all super classes of a given class are visited, but without producing additional columns for inherited attributes. In Chapter 7, it is shown how this failure is detected by using the previously presented contract and how it may be fixed. For this purpose, the next section shows how to use the consistency checking mechanisms of QVT Relations to verify PAMOMO contracts.

3.5 Operationalization of Contracts: From PaMoMo to QVT Relations

In order to use PAMOMO contracts as oracles, they have to be made operational. For this purpose, contracts are compiled into checkonly QVT Relations transformations and it is checked if they hold for certain models, according to the semantics shown in Section 3.2.1. In case a certain relation does not hold, the QVT engine provides information on which contract failed due to which bindings (i.e., bound objects, values and links). Three QVT transformations are generated: one containing the generated code for the preconditions, another one for the invariants, and the last one for the postconditions. In the following each one of them is detailed by providing a schematic template of the generated code and a concrete example. Since preconditions and postconditions offer many similarities they are dealt with first in a common subsection before it is separately dealt with invariants in the subsequent subsection.

3.5.1 Compilation of Preconditions and Postconditions

Compilation Scheme of Preconditions. Precondition patterns have an empty target compartment whereas postconditions have an empty source compartment since they both are specified over a single metamodel only. However, in QVT Relations all transformations must have at least two domains[2], but it is possible that these two domains conform to the same metamodel, i.e., a *pseudo domain* is introduced. Thus, in the case of pre- and postconditions, transformations with two domains conforming to the same metamodel are generated, which are actually bound to the same model. Fig. 3.19 shows the compilation scheme for positive and negative preconditions. In both cases, one top relation is produced with two domains (named `Source1` and `Source2` in the figure), bound to the same metamodel, as preconditions act on one single model.

If the resulting transformation is executed in check-only mode in the direction `Source1`→`Source2`, for each occurrence of the source of each top relation, the engine has to find an occurrence of the target of the relation to consider that the relation holds. For positive preconditions, one element, which always needs to be found, which is the root node of the precondition's object graph, will be added to the domain `Source1`. The full object graph will be added to the domain `Source2` only. Furthermore, in the *where* clause inequalities ensuring that two objects with compatible type may not be bound to the same object in the model are included, as well as the OCL constraint expression `EXP` of the precondition.

[2]At least by the used engine ModelMorf [149], which has been chosen since it is currently the only one that supports the check-only mode.

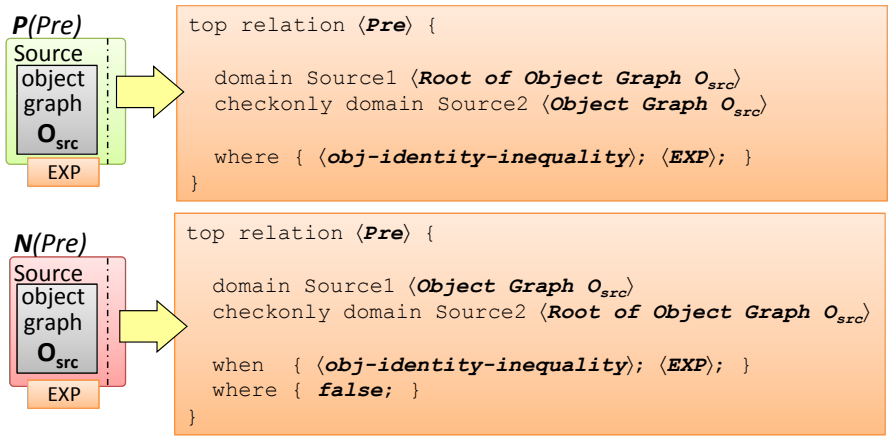


Figure 3.19: Compilation Scheme for Preconditions

Regarding negative preconditions, they demand the absence of an object graph. In this case, the object graph is added in the `Source1` domain, and the OCL constraint is included in the *when* clause. Moreover, as a negative precondition has to fail whenever the object graph is found, *false* is added to the *where* clause of the relation. Thus, finding the object graph in the source domain makes the relation fail because of the *where* clause.

Example. Fig. 3.20 shows a negative precondition taken from Fig. 3.7 and the generated QVT Relations code. The source object graph of the negative precondition pattern is compiled into the object graph for the `Source1` domain, whereas the `Source2` domain includes only the root node of this graph. In addition, three constraints are added to the *when* clause. The first two check that different objects in the relation are bound to different objects in the model. This is checked by inequalities in the identifiers of objects with same type in the `where` clause. Since

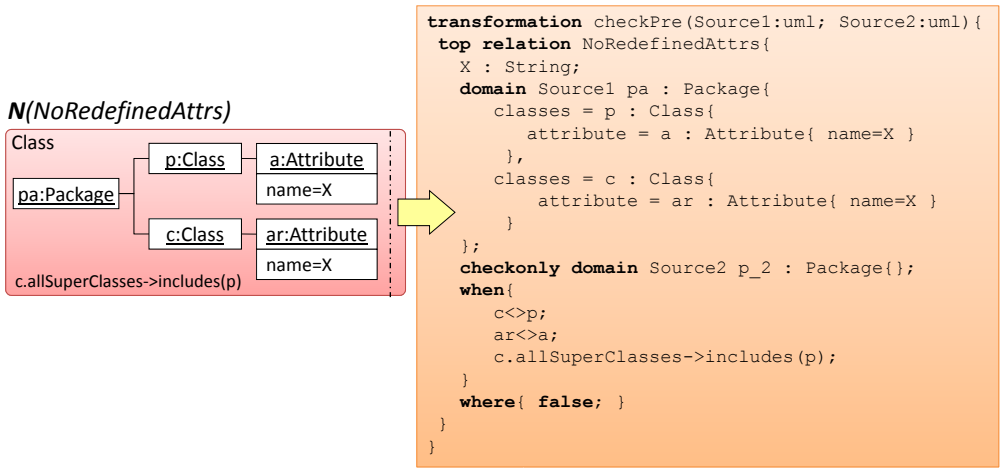


Figure 3.20: Compiling a Negative Precondition into QVT Relations

QVT Relations supports OCL, according expressions in patterns may be directly taken without changes in QVT Relations. Thus, the third constraint checks if the class `p` is a superclass of class `c`. Finally, the *where* clause includes the *false* statement, to make the relation fail in case a match for the source graph is found in the model.

Compilation Scheme of Postconditions. Fig. 3.21 shows the scheme of the compilation of positive and negative postconditions. Positive postconditions demand an occurrence of the target object graph, while negative postconditions are satisfied if there is no occurrence in the target object graph. Thus, the code generated from postconditions is similar to the one generated from preconditions but acting on the target metamodel instead.

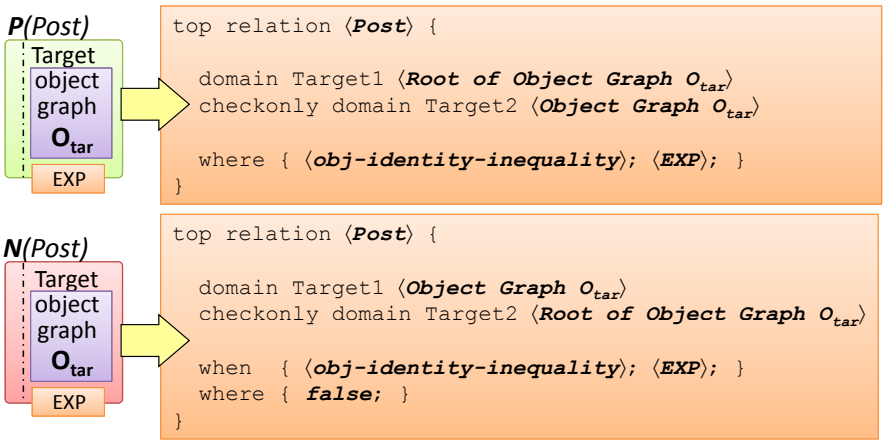


Figure 3.21: Compilation Scheme for Postconditions

Example. Fig. 3.22 depicts the negative postcondition shown in Fig. 3.7 and its compilation into QVT Relations. Please note that the resulting code is analogous to the code produced for the negative precondition example in Fig 3.20.

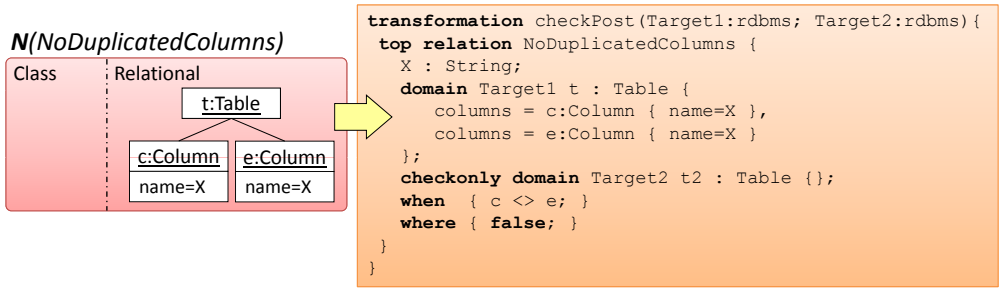


Figure 3.22: Compiling a Negative Postcondition into QVT Relations

3.5.2 Compilation of Invariants

Compilation Scheme. Fig. 3.23 shows the scheme of the compilation of positive and negative invariants. The scheme for positive invariants is similar to the one for preconditions and post-conditions, but now the two domains are typed on different metamodels and contain different object graphs. Moreover, the *when* clause includes the terms of the OCL invariant expression containing only elements of the source graph, whereas the remaining terms of the expression are added to the *where* clause.

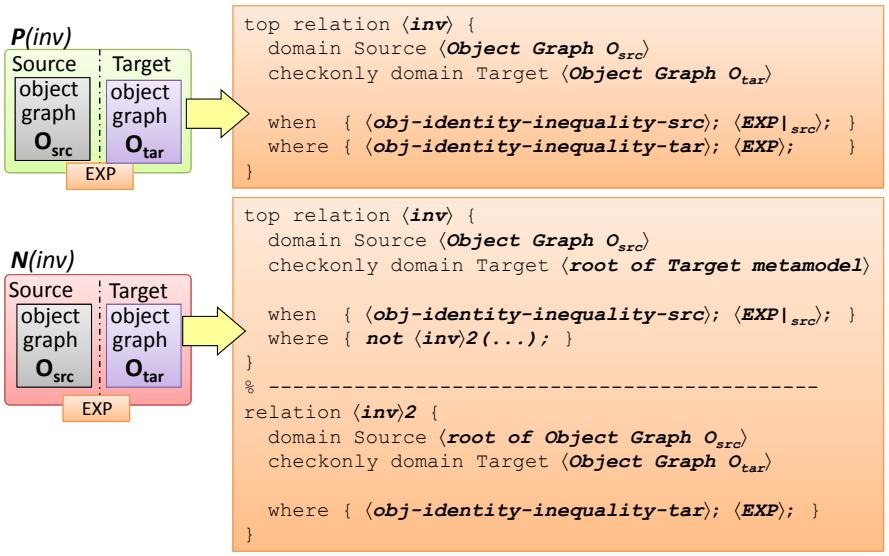


Figure 3.23: Compilation Scheme for Invariants

Another difference to the previous compilations is that negative invariants are split into two relations: the first one is top and looks for occurrences of the source, and the second one is non-top and looks for occurrences of the target when it is invoked from the *where* clause of the top relation. In this way, the top relation checks that for each occurrence of the source graph there is no occurrence of the target graph. This is latter checked by invoking the non-top relation in the negated *where* clause. Note that generating a single relation with a *false* statement in the *where* section, as done for negative pre- and postconditions (cf. Figs. 3.19 and 3.21), is not enough in this case. The reason is that such a relation fails if it does not find the complete target graph, however the relation should fail only if it does find both, the source and target graphs.

Example. Fig. 3.24 shows the compilation of the positive invariant modeling requirement 4 in Fig. 3.6. The generated relation has one domain for the source object graph and another domain for the target object graph. Its *when* clause includes an inequality to avoid binding the two classes `p` and `c` to the same object in the model, as well as the OCL constraint in the invariant as it only includes source objects. An example for the compilation of negative invariants is illustrated in the following subsection.

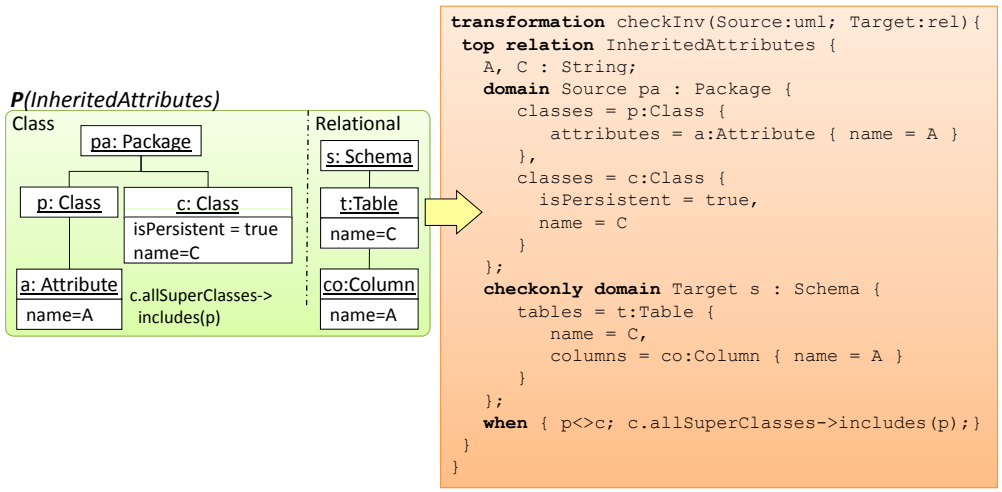


Figure 3.24: Compiling a Positive Invariant into QVT Relations

3.5.3 Compilation of Enabling and Disabling Conditions

Compilation Scheme. Enabling conditions are translated into top relations, which are checked in the *when* clause of the relation derived from the pattern they constrain. If the relation derived from the enabling condition does not hold, the relation derived for the pattern vacuously holds. This compilation scheme is shown in Fig. 3.25. For disabling conditions the scheme is the same, but they are invoked in the *when* clause preceded by "not". If a pattern contains several disabling conditions, their invocations are concatenated with a logical "and".

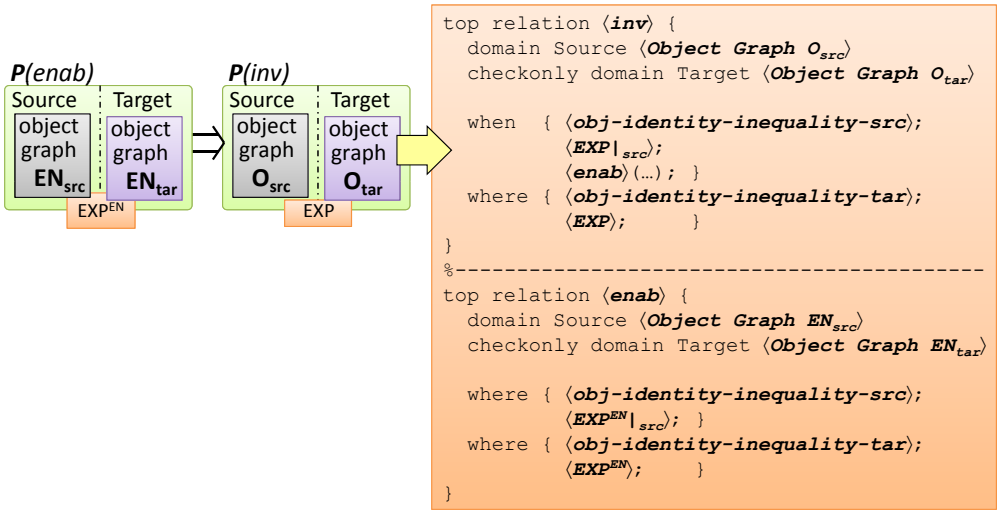


Figure 3.25: Compilation Scheme for Enabling Conditions

Example. Fig. 3.26 shows the code generated for the negative invariant of Fig. 3.11, which has an enabling condition. In particular, the relations `NoTableForTransientClass` and `NoTableForTransientClass2` are generated from the negative invariant, and `PackageAndSchema` from the enabling condition. Hence, top relation `NoTableForTransientClass` only needs to hold for a particular `Package` and `Schema` when they satisfy the relation `PackageAndSchema`, which is checked in the *when* clause.

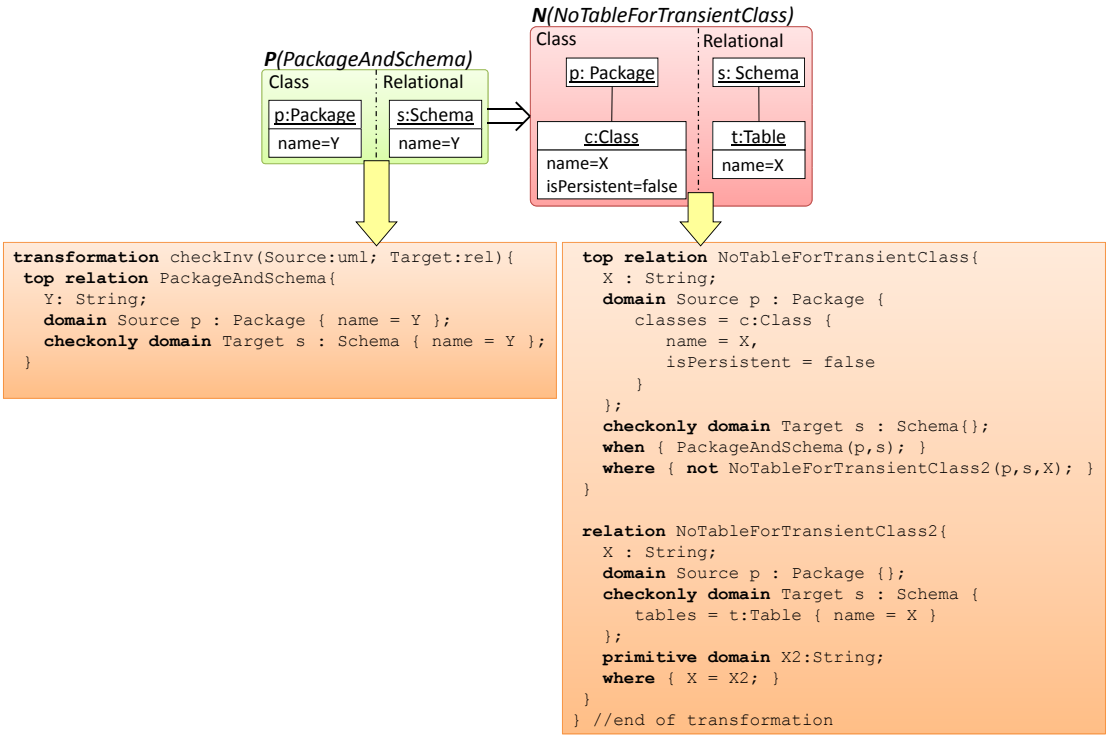


Figure 3.26: Compiling an Enabling Condition for a Negative Invariant into QVT Relations

In this example, the relation `NoTableForTransientClass` invokes `NoTableForTransientClass2` passing the string variable `X` as a parameter, which has to be defined as a primitive domain in the invoked relation. Moreover, due to a limitation of the used QVT Relations engine (ModelMorf [149]), which only supports relations with two domains, the compilation of enabling conditions containing more than one object in the source or target requires special treatment. This is so as any invocation of a relation must receive exactly two objects as parameters, plus any number of primitive values. Thus, if the enabling condition contains several objects in the source or the target, all objects should be assigned to according parameters during invocation, which is not allowed. This problem is solved by passing the object identifiers (which have a primitive type, i.e., string, and may therefore be passed as primitive domains) instead of the objects themselves.

3.5.4 Compilation of Sets

Compilation Scheme. QVT Relations allows matching for collections of objects (sets, bags or sequences) using so-called *collection templates*. The ModelMorf QVT engine provides two kinds of collection templates: (i) *enumerations* for the extensional definition of sets, and (ii) *comprehensions* for its intensional definition. Enumerations match for a certain number of members in a collection. For instance, `classes = pclasses : Set(Class) {c1, c2 ++ _ }` matches for two classes in the reference `classes`. The underscore is a wildcard which matches for the rest of the collection. Comprehensions allow matching members in a collection using a condition. For instance, `classes = pclasses : Set(Class) {} {pclasses->forall (c | c.isPersistent)}` matches all persistent classes in the reference `classes`.

As Fig. 3.27 shows, sets in PAMOMO patterns are compiled into collection templates. Enumerations are generated if the elements in the set are not constrained by any condition, and comprehension otherwise. As before, the OCL expressions using only source variables and source set variables are included in the *when* clause, whereas the rest are included in the *where* clause.

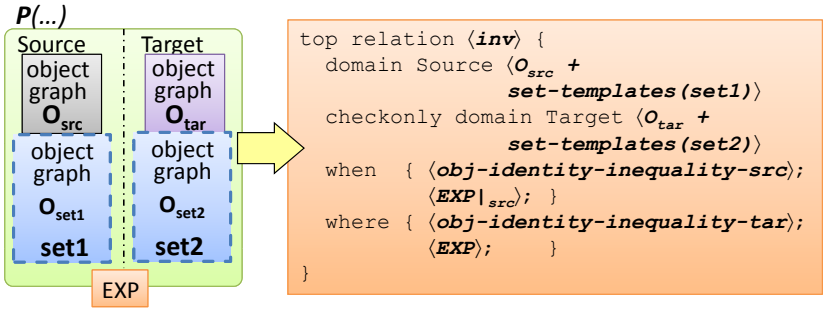


Figure 3.27: Compilation Scheme for Sets

Example. Fig. 3.28 lists the code generated from the invariant with sets shown in Fig. 3.15. The set `pclasses` is translated into a comprehension because it contains a condition matching for persistent classes only (`isPersistent = true`). In contrast, the set `tabs` is compiled into a simple enumeration. The OCL expression is added to the *where* clause of the relation because it relates set variables of the source and target. This expression fails if the number of persistent classes is not equal to the number of tables.

If a set contains an arbitrary graph having more than one element, then one additional relation is generated looking for occurrences of this graph structure. This relation is used to filter which elements should be added to the collection (i.e., only those preserving the relation).

3.5.5 Summary of the Compilation

Table 3.4 summarizes the compilation of PAMOMO contracts into QVT Relations code. This section has shown that PaMoMo contracts may fully be expressed in terms of checkonly QVT

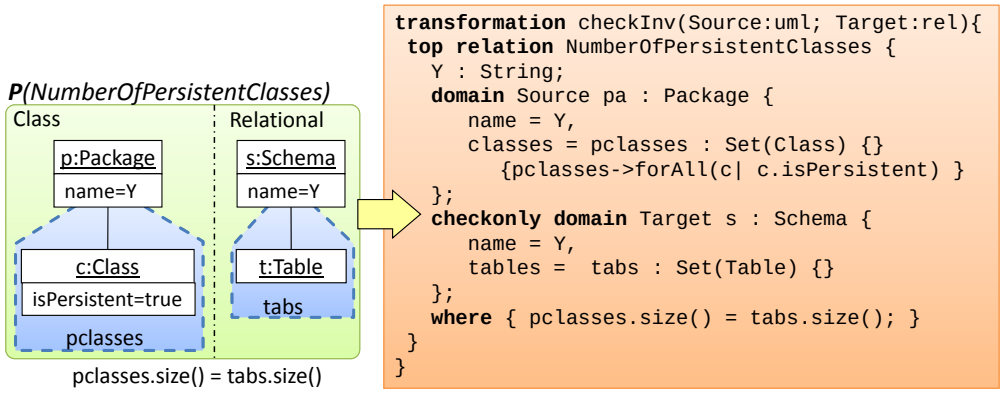


Figure 3.28: Compiling a Positive Invariant with Sets into QVT Relations

Relations offering a more compact specification of contracts than the direct use of QVT Relations. This is due to the availability of different kinds of patterns (positive and negative invariants, pre- and postconditions) and features (enabling/disabling conditions, sets) of PAMOMO. At the same time, it has to be emphasized that it is not the aim of QVT Relations to specify transformation contracts but rather to specify model-to-model transformations. Furthermore, using PAMOMO for contract specification reduces the effort and the number of potential errors in comparison to directly using QVT Relations.

Table 3.4: Summary of PAMOMO-to-QVT Compilation

PaMoMo Concept	QVT Relation Representation
P(Pre/Post)	1 relation with pseudo domain
N(Pre/Post)	1 relation with pseudo domain + **false** in where clause
P(Inv)	1 relation
N(inv)	2 relations + negated call of relation in **where** clause of relation 1
Enabling condition	1 relation + call of the relation from **when** clause of relation for constrained pattern
Disabling condition	1 relation + negated call of the relation from **when** clause of relation for constrained pattern

3.6 Executing PaMoMo Contracts

In the previous sections it was discussed how to formalize transformation requirements using PaMoMo contracts, which may be made executable by means of QVT Relations in order to test a certain transformation, i.e., whether the transformation fulfills the posed requirements or

not. This section now elaborates on the actual execution of contracts and shows how model transformations may be tested, exemplified by means of the running example introduced in Section 1.2. In this respect, Fig. 3.29 repeats the input model and depicts the target model generated by the transformation under test as well as the verification log, stating which contracts succeeded and which contracts failed. Additionally, traces to the actual model elements that caused the contract to fail are given, which is detailed in following.

As a first step of the execution, the preconditions are evaluated on the source model. The test source model depicted in Fig. 3.29 fulfills the requirement five of the running example, i.e., it contains no redefined attributes (cf. first line in verification log). Since requirement 5 is the only precondition, all preconditions are fulfilled and therefore the actual transformation may be executed in order to achieve a target model. A potential solution of the transformation of the running example was presented in Section 3.4 using QVT Relations. If this transformation is executed, the target model depicted in Fig. 3.29 is generated. In order to test if specified transformation is correct, in a second step, invariants are executed, checking if the target model generated by the transformation fits to the expected (parts of the) target model of the invariants, e.g., in the example requirements 1 to 4 are checked since they were expressed as corresponding invariants. When executing the specified invariants, one may see that only requirements 1 to 3 are satisfied but not requirement 4 (cf. verification log in Fig. 3.29).

On inspecting the generated target model in Fig. 3.29 one may realize that the transformation specified in Fig. 3.18 produces a `Schema s1` named `University` which stems from the `Package p1` (cf. test input model in Fig. 3.29), checked by the first invariant. Additionally, the second invariant checks if persistent classes are translated into equally named tables, which is also true since two according tables have been created, i.e., only the tables named `Student` and `Professor` have been created since their according classes are persistent, but no table named `Person` has been created since the according class is not persistent. Furthermore, every

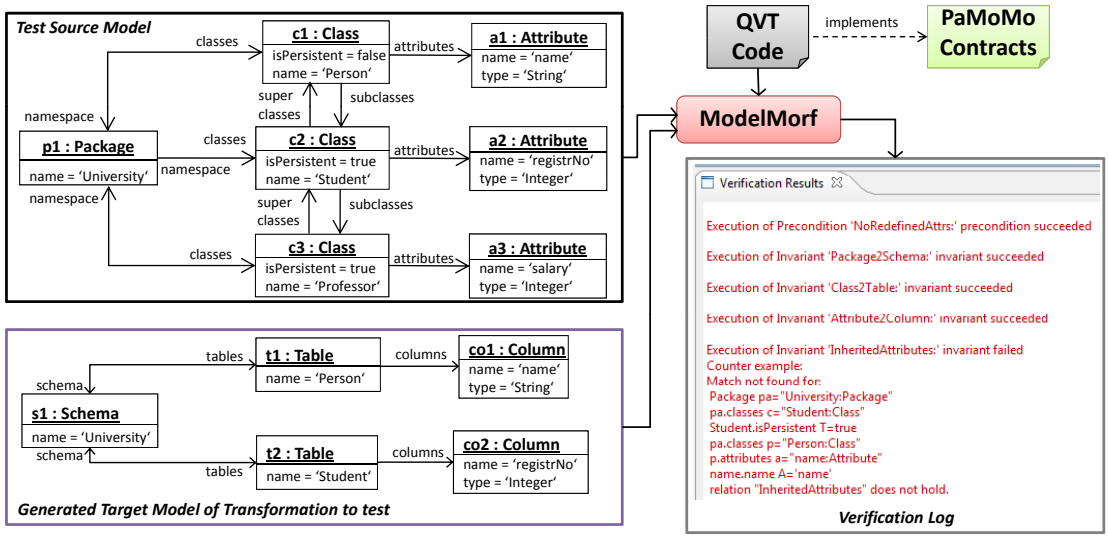


Figure 3.29: Verification Results of Requirements 1-4 of Running Example

direct attribute, i.e., `registrNo` in case of class `c2` and `salary` in case of class `c3`, has been correctly transformed into `Column` instances linked to the corresponding table, as demanded by invariant 3. Nevertheless, invariant 4 fails which is due to the fact that for the `Package` named `University` and the `Class` named `Student` with the attribute `name` – included in superclass `Person` – no corresponding `Column` in the `Table` named `Student` can be found. It may be concluded that the implementation of the transformation in Fig. 3.18 does not handle inherited attributes appropriately. For a more detailed discussion by applying the implemented prototype the reader is referred to Chapter 8.

As may be seen from this example, the specified contracts helped the transformation designer to detect that there is a failure in the specification, i.e., contracts are useful in observing facts according to [173]. Since PaMoMo contracts are independent from the underlying transformation languages, there is no direct relationship between contracts and the actual transformation rules. Therefore, a mechanism is needed to (i) first identify the rules that caused the error and (ii) to find the origin of the defect, i.e., often a defect is introduced already earlier and caused dependent rules to to fail. The verification log in Fig. 3.29 states which source elements caused a certain contract to fail. This information could be used to execute the transformation with these input elements in order to find the defect. In order to support the transformation designer in this task, appropriate debugging facilities are required. Nevertheless, current transformation languages and their underlying transformation engines hardly provide any debugging mechanisms in order to reduce the effort in finding the defect. In this respect, the following Chapter 4 introduces a runtime model for model-to-model transformations which provides an insight into the actual execution of the transformation and provides the basis for sophisticated debugging facilities which are presented in Chapter 7.

3.7 Summary

In this section a declarative, formal, visual language to specify behavioral semantic contracts for model-to-model transformations has been proposed which is called PaMoMo. In this way, PaMoMo may be used to specify *preconditions* and *postconditions* to express that an input or output model should or should not contain certain configurations of elements. Furthermore, they may be used to specify *invariants*, i.e., what conditions need to be satisfied by any pair of input/output models of a transformation. In order to execute the specified contracts, they are translated to QVT Relations which are then executed in check-only mode in order to check if the relation holds for the candidate models, resulting in a verification log. This verification log shows traces to the model elements that caused the relation to fail. This information may then be used for debugging the model transformation, which is the focus of the following chapters.

Chapter 4

Transformation Nets - A Runtime Model for Model Transformations

No great discovery was ever made
without a bold guess.

— Sir Isaac Newton

Contents

In order to support the transformation designer in finding the origin of a defect, this chapter introduces the fundamentals of the Transformation Net formalism, which is a Domain Specific Language (DSL) on top of Colored Petri Nets (CPNs) [72]. Transformation Nets serve as a runtime model for the execution of model transformations, making its execution semantics explicit in order to foster debuggability. After introducing the general idea, first, the representation of metamodels and models in Transformation Nets is discussed followed by the transformation logic itself. Additionally, it is shown how conditions and functions may be expressed in Transformation Nets and how the presented concepts interact together in order to specify a model-to-model transformation. Finally, modules are discussed in order to provide modularization concepts in Transformation Nets.

4.1 Transformation Nets at a Glance

The basic idea of runtime models is to reason about the operating environment and the runtime behavior of systems as well as to provide appropriate abstractions from code-level details of the applications at runtime [95]. Consequently, a runtime model for model transformations should not only provide means to represent the specified transformation logic as a model, as e.g., proposed in [17], but should also incorporate information about its actual execution, i.e., it should make the operational semantics of a model transformation explicit. This section introduces the general idea of the Transformation Net formalism, which provides a runtime model for model-to-model transformations. Transformation Nets thereby form a DSL on top of Colored Petri Nets (CPN) [72]. CPNs extend the basic concepts of Petri Nets [118, 124] by the possibility of attaching data to tokens. Petri Nets in general have been chosen in order to profit from their runtime model and their formal semantics. On the one hand Petri Nets allow for abstraction from control flow, which is prevalent in declarative transformation approaches. This is achieved since transitions may fire autonomously, depending on the markings contained in the places, only. On the other hand, also the statefullness of imperative approaches is preserved, since the actual state of execution is represented by the current available tokens. CPNs as a special form of Petri Nets have been chosen since they allow the tokens to carry data, which are called token colors and are thus able to represent an actual model. By providing an inscription language, the data values may be queried and accordingly modified when firing a transition. In order to hide low-level details and circumventing restrictions of CPNs with respect to model transformations, Transformation Nets as a DSL on top of it, will be presented in the following.

The conceptual architecture of Transformation Nets is pictured in Fig. 4.1, showing a source metamodel on the left hand-side and a target metamodel on the right-hand side. Furthermore, an input model conforming to the source metamodel, as well as an output model conforming to the target metamodel that represents the output of the transformation is depicted. In between, the transformation logic resides, describing the correspondences between the metamodel elements. These common parts of model transformations have to be described by means of CPN concepts. The middle of Figure Fig. 4.1 shows a Transformation Net, which represents the *static parts* of

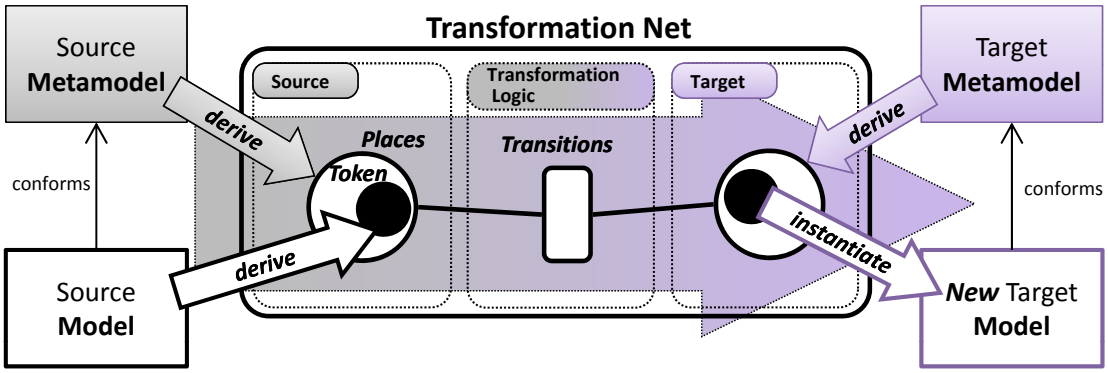


Figure 4.1: Conceptual Architecture of Transformation Nets

the transformation (i.e., metamodels and models) as places and tokens, respectively and the *dynamic parts* (i.e., the transformation logic) as according transitions. In this respect, Transformation Nets provide an explicit, integrated representation of common concepts of model-to-model transformations.

A first version of Transformation Nets has already been presented in [125]. Nevertheless, in the course of this thesis further development of the runtime model has been considered, going beyond the contributions proposed in [125]. In a first step, the common core concepts of model-to-model transformation languages are analyzed in order to systematically integrate the concepts in the runtime model. This leads to numerous extensions in the underlying metamodel, e.g., besides the more explicit representation of metamodels and models (specific types have been introduced as discussed in the following), also the specification of functions and conditions has been included (cf. Subsection 4.4.2). Additionally, a focus is set on representing modularization and reuse concepts, e.g., modules and rule inheritance, which will be discussed in detail in Chapter 5. Furthermore, a formal basis is provided by its full compilation into CPNs, which will be discussed in Chapter 6. In this respect, Transformation Nets can be used for three different purposes. First, they can be used as a transformation language itself, especially tailored to reuse of transformation logics as presented in [125]. Second, Transformation Nets may serve as a target language to which other high-level transformation languages may be compiled in order to benefit from the provided runtime model and the integrated debugging features, which is the focus of this thesis. Finally, Transformation Net may also serve as runtime model for a new transformation language, as discussed in Subsection 9.2.3.

4.2 Core Concepts of Transformation Nets

Since Transformation Nets are intended to provide a runtime model for model-to-model transformations, first, the common features of existing transformation languages, which need to be accordingly supported by Transformation Nets, have to be identified. Currently, numerous transformation languages are available (cf. [38] for an overview), which follow different paradigms, i.e., declarative, imperative, or hybrid. Since imperative languages and also hybrid ones require the transformation designer to specify the actual execution of model transformations, e.g., by using control statements like conditions or loops, the actual execution thereof may be followed by debuggers that allow for stepwise execution of the code as known from traditional programming languages. The situation is different for declarative transformation languages, since only *what* should be transformed has to be specified but not *how* this is done, which is typically hidden by an according execution engine. The goal of Transformation Nets is to make this execution explicit and thus, Transformation Nets focus on providing an explicit runtime model for declarative, rule-based model-to-model transformation languages, in a batch and exogenous scenario, i.e., a source model is transformed into a new target model.

To identify the required concepts, (i) the features of declarative, rule-based model-to-model transformation languages, and (ii) the classification of model transformation approaches presented in [38] have to be analyzed. The identified features are expressed in terms of a metamodel (shown in Fig. 4.2) which illustrates the core concepts of transformation languages, building the basis for Transformation Nets. Since the focus is on rule-based model-to-model transformation

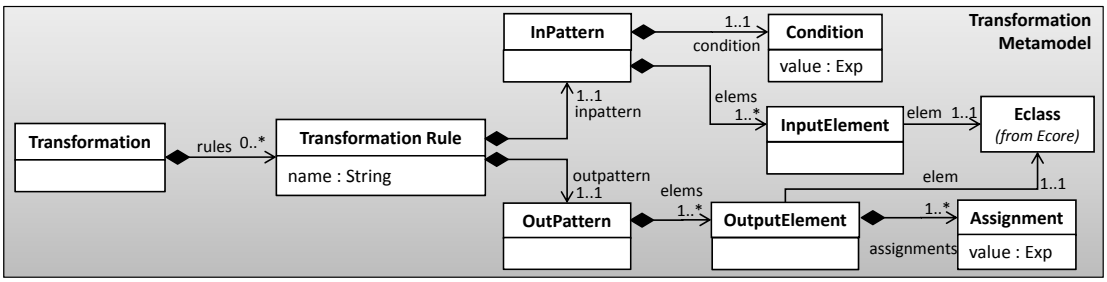


Figure 4.2: General Concepts of Transformation Languages

languages, a `Transformation` specification typically consists of numerous `TransformationRules`. These transformation rules need to provide means to specify some relationships between elements of the source model and elements of the target model that is created. Therefore, `TransformationRules` include an `InPattern`, referring to `InputElements` of the source metamodel, and an `OutPattern`, referring to `OutputElements` of the target metamodel (cf. references `InputElement.elems` and `OutputElem.elems` in Fig. 4.2). A first general distinguishing criterion is the allowed number of input and output elements. Several transformation languages allow to match for a combination of several input elements, e.g., ATL [73] or TGGs [83], whereas others restrict themselves to a single input element, e.g., ETL [81]. On the one hand, matching only a single input element tends to be more efficient in terms of execution time, since no potential combinations of input elements need to be calculated. On the other hand, this often requires to specify more complex OCL expressions in order to navigate through the metamodel. Furthermore, transformation languages typically support the definition of a `Condition` in order to filter certain model elements, most often being specified with OCL. Finally, transformation languages provide the possibility of setting the values for target features by means of `Assignments`. It is important that the features are contained in the according `EClasses` which are referred by the corresponding `InputElements` and `OutputElements`, respectively.

These common concepts of transformation languages need to be accordingly represented in Transformation Nets. The abstract syntax of the Transformation Net DSL is formalized by means of a metamodel (cf. Fig. 4.3) conforming to the Ecore[1] meta-metamodel, the Eclipse realization of OMG's MOF[2] standard. On the one hand, the Transformation Net metamodel is based on the concepts presented in the metamodel in Fig. 4.2 and, on the other hand, it is based on CPN concepts [72], which are adapted to the special requirements, occurring in the domain of model-to-model transformations. In particular, in order to be able to encode metamodels, which corresponds to the concept of `InputElements` and `OutputElements` in transformation languages, different kinds of places (cf. Section 4.3) are introduced, represented by the package `StaticElement` (cf. Fig. 4.3). Additionally, also the model elements, which are typically not represented in current transformation languages, are explicitly represented by means of tokens.

[1] http://www.eclipse.org/modeling/emf/?project=emf
[2] http://www.omg.org/mof

The second major adaptation concerns transitions. Since transitions are used to realize the actual transformation logic, a well established specification technique from graph transformations [41] is adapted, which describe their transformation logic as a set of graphically encoded productions rules (cf. Section 4.4). These are represented by concepts of the package `DynamicElement`. `Transitions` represent `TransformationRules`, whereas different kinds of patterns fulfill the task of `InPatterns` and `OutPatterns`, as detailed in Section 4.4. The package `Connectors` is responsible for connecting the static parts with the dynamic parts. Finally, the package `Container` aggregates the presented concepts, i.e., containers for the source and target metamodels and the transformation logic are provided.

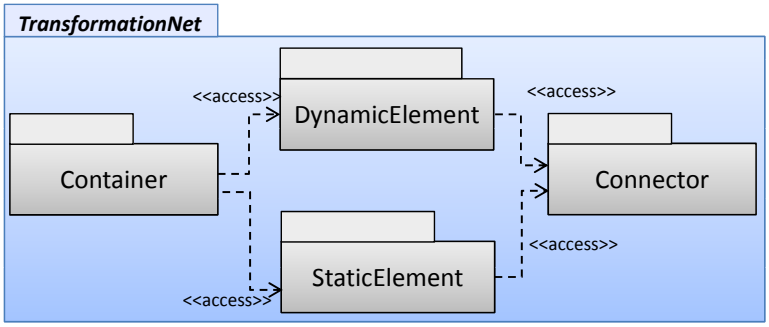


Figure 4.3: Packages of the Transformation Net Metamodel

4.3 Static Parts of Transformation Nets

When employing Transformation Nets, in a first step, the static parts of a model transformation, i.e., metamodels and models, need to be represented in the formalism, as can be seen in Fig. 4.4. In this respect, it is first detailed, how object-oriented metamodels, e.g., metamodels based on Ecore, and their conforming models can be represented in Transformation Nets. Second, the scope is broadened to other data models as well, e.g., XML Schemas and Ontologies, discussing how concepts of these data models relate to Ecore concepts and how they may be represented in Transformation Nets.

4.3.1 Representing Object-Oriented Metamodels in Transformation Nets

In MDE, most of today's transformation languages allow to specify transformations between object-oriented metamodels (M2), which themselves conform to a meta-metamodel (M3), e.g., Ecore or MOF, in order to transform according models (M1). In today's transformation languages both metamodels, i.e., source and target metamodel, must conform to the same meta-metamodel. In order to explicate the translation of concepts in object-oriented metamodels to Transformation Nets, the Ecore-based `Class` and `Relational` metamodels of the running example are used in the following. First a short overview on the concepts available in Ecore is given.

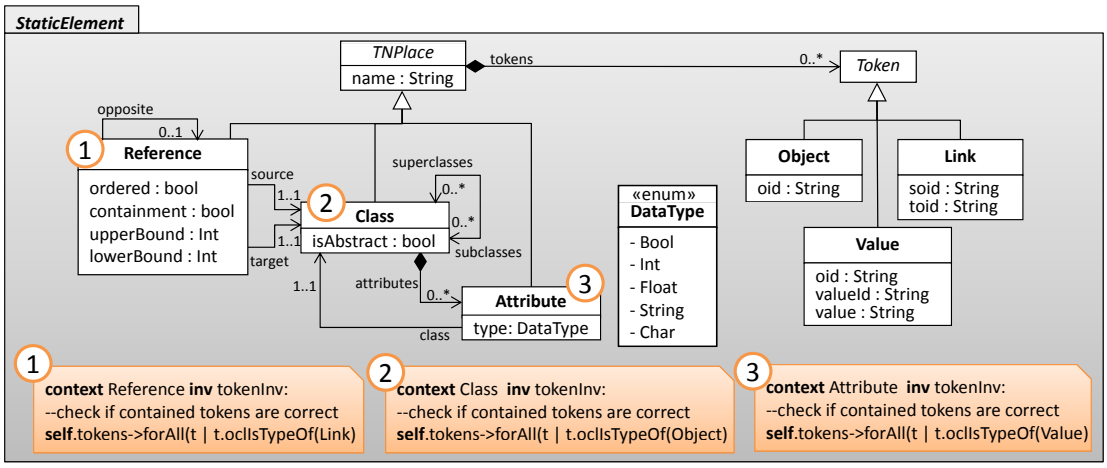


Figure 4.4: Static Elements of Transformation Nets

4.3.1.1 The Ecore Meta-Metamodel

As already mentioned, Ecore, being the Eclipse realization of OMG's MOF standard, has reached wide-spread adoption in practice and is part of the Eclipse Modeling Framework[3] (EMF). The Ecore meta-metamodel (cf. Fig. 4.5) is used to define metamodels, but also provides means to generate a Java implementation of the metamodel. Since the focus of this thesis is on model transformations between metamodels, only the concepts of Ecore that are needed for metamodeling, namely `EClass` (for representing classes), `EAttribute` (for representing attributes) and `EReference` (for representing references), are considered. Although data types, represented by the class `EDataType` and enumerations, represented by the classes `EEnum` and `EEnumLiteral` are used in metamodels, they need not be represented in Transformation Nets explicitly. This is due to the fact that they may be treated equal to data values, i.e., on a conceptual perspective there is no difference between an enumeration and an integer standard data type, thus, none of them are explicitly represented, instead only their concrete values are taken into account. Furthermore, please note that only single-valued attributes are considered. Finally, concepts provided for code generation (e.g., `EFactory`) and for behavioral modeling (e.g., `EOperation` or `EParameter`), as well as annotations (`EAnnotation`) are not represented in Transformation Nets.

4.3.1.2 Representing Metamodel Elements in Transformation Nets

The representation of the involved metamodels in a model-to-model transformation scenario requires a conversion from the graph-based paradigm underlying Ecore into the set-based paradigm underlying Petri Nets. The design rationale behind this translation is to rely on the core concepts of an object-oriented meta-metamodel, i.e., the graph, which represents the metamodel, consists of *classes*, *attributes*, and *references*. These metamodel elements are translated into according

[3] http://www.eclipse.org/emf

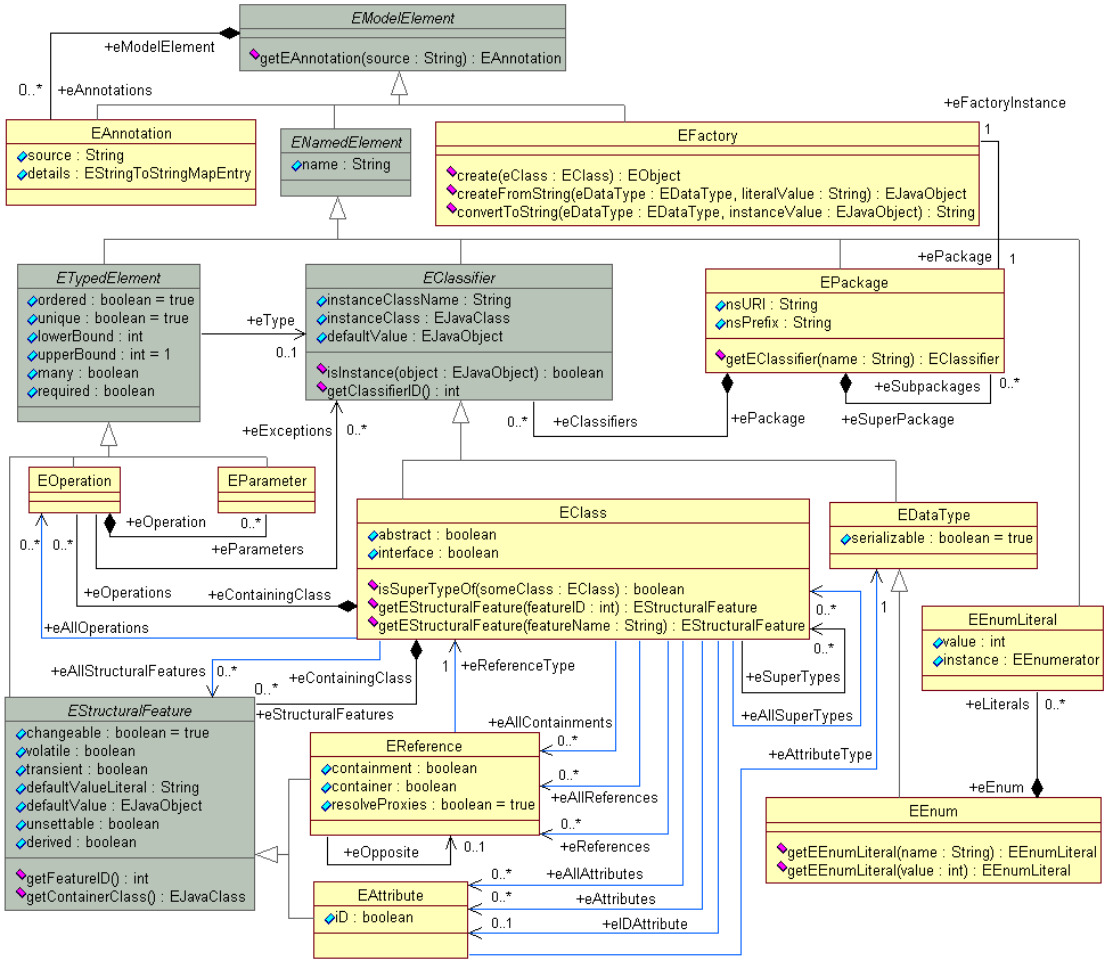


Figure 4.5: The Ecore Meta-Metamodel [40]

subtypes of `TNPlace` in Transformation Nets. Fig. 4.6 shows an overview of the proposed translation, whereby an extract of the source metamodel of the running example is used, represented in their respective concrete and abstract syntax. In order to minimize the visual gap between class diagrams (which are usually used to represent metamodels graphically) and Petri Nets, Transformation Nets try to visually combine these two representations by providing an extra compartment in terms of an oval, which is typically used to depict places in Petri Nets, as can be seen in Fig. 4.6(d)

Representation of Classes. Both, abstract as well as concrete classes (i.e., all instances of `EClass`) are translated into `Class` instances in Transformation Nets. `Classes` may be set abstract (cf. boolean attribute `Class.abstract` in Fig. 4.4) and are allowed to inherit from one or more superclasses (cf. reference `Class.superClasses` and its opposite reference `Class.subClasses` in Fig. 4.4), which is represented in Ecore in terms of the reference `EClass.eSuperTypes`. Fig. 4.7 shows the translation of the complete source metamodel of

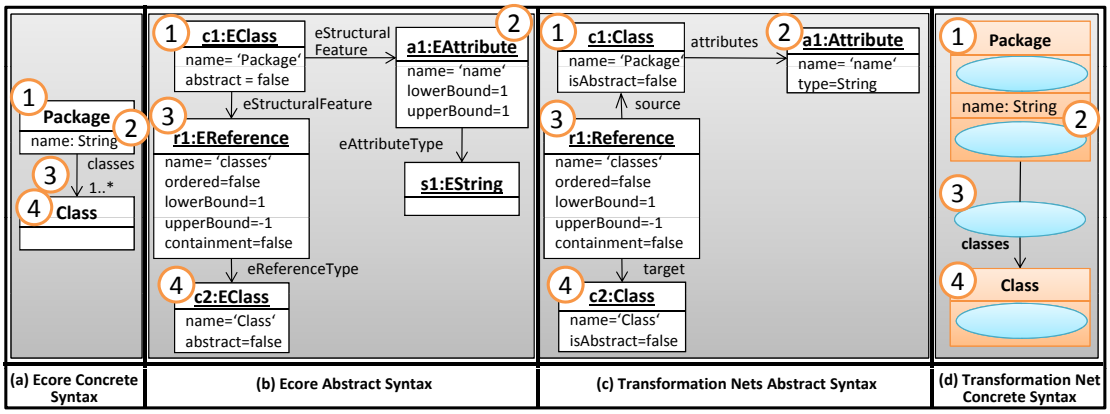


Figure 4.6: Representation of Metamodel Elements in Transformation Nets

the running example into Transformation Nets, i.e., the classes `ModelElement`, `Package`, `Class` and `Attribute` are represented by according places.

Representation of Attributes. Similar to classes, all instances of `EAttribute` are translated into `Attribute` instances (cf. Fig. 4.4 and Fig. 4.6(d)). The attribute `Attribute.type` stores the according datatype by means of the enumeration `DataType`, which provides the available standard datatypes (`Bool`, `Int`, `Float`, `String`, `Char`). Again, Fig. 4.7 shows the translation of the attributes of the source metamodel of the running example into Transformation Nets, i.e., the attributes `ModelElement.name`, `Class.isPersistent` and `Attribute.type` are represent by according nested places.

Representation of References. References in the involved metamodels (i.e., all instances of `EReference`) are translated into `Reference` instances in Transformation Nets (cf. Fig. 4.6(d) and Fig. 4.4). To not loose any information in the translation process, `References` provide equal attributes and references as `EReferences` do, i.e., `ordered`, `containment` (to specify aggregations), `lowerBound`, `upperBound` and `opposite` (note, that the opposite reference is used in Ecore to express bidirectional associations). Fig. 4.7 shows the representation of the references of the source metamodel of the running example.

4.3.1.3 Representing Models in Transformation Nets

The graph, which represents a model conforming to a certain metamodel, consists of *objects*, *values* and *links* which have to be accordingly translated into `Tokens` (cf. Fig. 4.9(a)) in Transformation Nets, which are then put into the according places (cf. Fig. 4.9(b)), which is specified by means of the OCL constraints shown in Fig. 4.4. For example, an `Object` token may only be contained in a `Class` place. Thus, Transformation Nets not only represent the involved metamodels, which are required to specify the transformation logic, but also the involved models in order to provide an explicit view on the execution of a model transformation.

Representation of Objects. For every object that occurs in a source model, an `Object` instance in Transformation Nets is produced, which is put into the place that corresponds to the respective class in the metamodel. The "color" of a token is in fact expressed by means

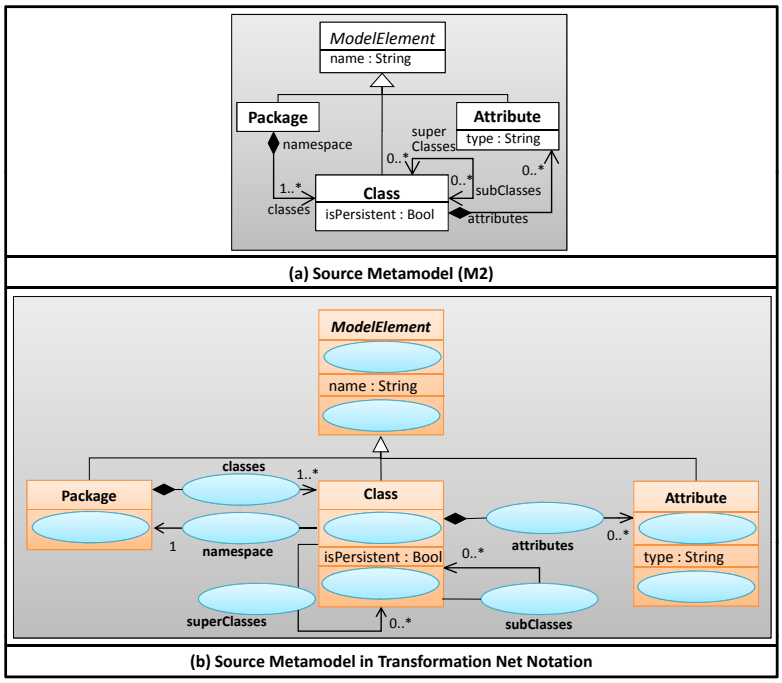


Figure 4.7: Source Metamodel Translated to Transformation Net

of a unique value that is derived from the identifying attribute of the original model object (cf. attribute `Object.oid` in Fig. 4.4 and Fig. 4.8). With respect to the running example, one may see that each instance of a class got represented through a respective `Object` token (cf. Fig. 4.9). Since the class `NamedElement` is abstract and therefore, no direct instances may exist, no token is put into the according place.

Representation of Values. For every value as an instance of an attribute, a `Value` token is produced. A value token is represented by an object id (cf. attribute `Value.oid`) of the owning object (upper part of the token) as well as a unique id for a certain value (cf. attribute `Value.valueId` and lower part of the token) for which again according colors are derived to provide a visual representation. The actual value of the attribute is stored as a string representation in the attribute `Value.value`, and is represented by the label in the lower part of the token

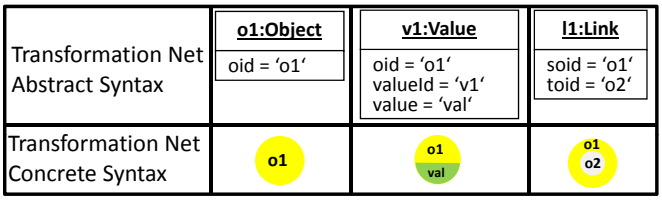


Figure 4.8: Overview on Concrete Syntax of Transformation Net

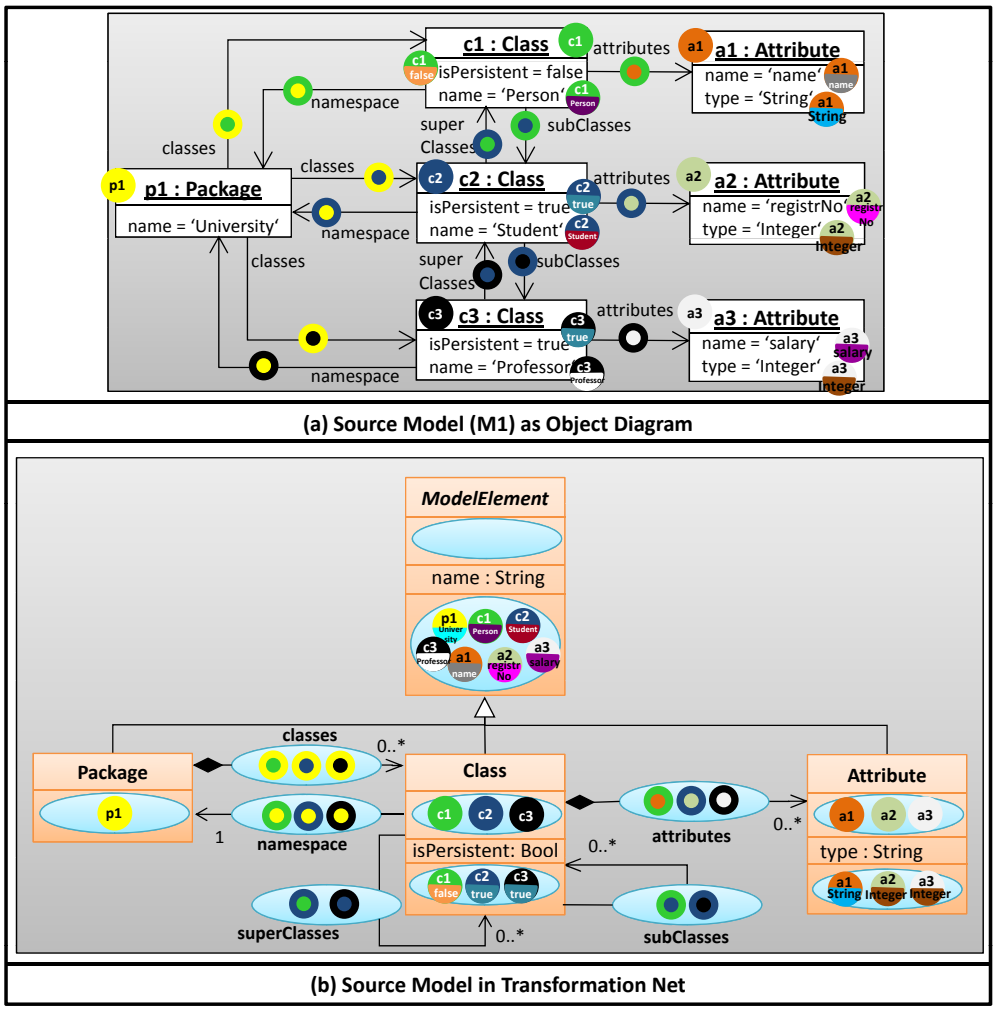


Figure 4.9: Source Model Translated to Transformation Net

(cf. Fig. 4.8). Please note that, if two attributes exhibit the same value, the same id is generated and thus, the lower parts of the tokens are equally colored (cf. e.g., the color for attribute value `true` of the tokens representing the attributes `Class.isPersistent` in Fig. 4.9).

Representation of Links. Finally, for every link as an instance of a reference, a `Link` token is produced. The `Link.soid` attribute of such a token (cf. Fig. 4.3) refers to the id of the token that corresponds to the owning object. The `Link.toid` is given by the id of the token that corresponds to the linked target object. Notationally, a link token is represented as a ring denoting the `Link.soid` color surrounding an inner circle denoting the `Link.toid` color (cf. Fig. 4.8). Concerning the example, one may see that for each link in the source model an according link token is generated. Therefore, e.g., the place representing the `Package.classes` reference contains three tokens, which represent the containment relationships of the `Class` instances `c1`, `c2`, and `c3` to the `Package` instance `p1`.

As discussed above, a source model may be translated into tokens, forming the initial marking of the Transformation Net. On executing the transformation by firing its transitions (cf. Section 4.4), tokens are generated into places that represent the target metamodel. These tokens are then serialized to an according graph structure again, such that a target model results that conforms to the target metamodel. Consequently, every `Object` token is translated to an according `EObject` in Ecore with its attributes and references set to the values derived from the according `Value` and `Link` tokens.

4.3.2 Going beyond Object-Oriented Metamodels

In the previous section, the focus was on translating object-oriented metamodels and their corresponding models into corresponding `Places` and `Tokens` in Transformation Nets. The concepts of metamodels and models also arise in other engineering domains. In case of data engineering, the role of metamodels is played by schemata (e.g., in the form of database schemata or XML schemata), and the role of models by corresponding instance data (e.g., database tuples or XML documents). In case of ontology engineering, the role of metamodels is played by ontological concepts (the so-called T(ype)-box) and the role of models by corresponding individuals (the so-called A(ssertional)-box). Metamodels, schemata or ontologies are themselves instances of certain meta-metamodels, e.g., Ecore, XML schema[4], or OWL[5] (cf. Fig. 4.10). Since these data models build upon common core concepts, as already stated in [67] and summarized in Table 4.1, the according concepts may also be translated to `Places` and `Tokens` in Transformation Nets by means of adapters (cf. Fig. 4.10). In the following, such adapters from XML schema and OWL to Transformation Nets are shortly described.

Table 4.1: Common Core Concepts in Different Meta-Metamodels

Common Core Concepts	Ecore	XML Schema	OWL	Transformation Nets
Class	EClass	<xs:complexType>	<owl:Class>	Class (place)
Attribute	EAttribute	<xs:attribute>	<owl:DatatypeProperty>	Attribute (place)
Reference	EReference	<xs:key>, <xs:keyRef>	<owl:ObjectProperty>	Reference (place)
Inheritance	eSuperTypes	<xs:extension base>	<rdfs:subClassOf>	Class.superclasses

XML Schema to Transformation Nets. The EMF framework already provides support for XML schemas and XML models, i.e., an XML schema might be automatically translated to an Ecore metamodel and XML models to according Ecore models whereby the details of the translation are explained in the EMF documentation[6]. In this respect, an adapter which directly translates XML schemas into Transformation Nets may follow this transformation. For example, in the EMF documentation it is described that `complexType` instances may be represented in Ecore by according `EClasses`. Since instances of `EClasses` are translated to `Class` places it is natural to directly translate instances of `complexType` to according `Class` places. Furthermore, `Elements` in XML schemas are mapped to according `Attribute` places in

[4] http://www.w3.org/XML/Schema
[5] http://www.w3.org/TR/owl-features
[6] http://www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf

case they are typed to standard datatypes or enumeration values. If `elements` are typed to complex types, they represent references and are thus translated to `Reference` places. Valid XML files may then be translated to according tokens. For further details, the reader is referred to the EMF documentation.

OWL to Transformation Nets. In [76] a translation from OWL concepts to Ecore has been presented. This proposed translation may be followed in order to provide an adapter which directly translates ontologies into Transformation Nets. In the following, the translation is shortly summarized. The OWL concept `OWLClass` basically corresponds to the concept `EClass` and may therefore be mapped to places similar to the way described before. Additionally, in an OWL ontology, classes may be marked to be equivalent, in order to specify synonyms. In contrast, Ecore allows only for distinct classes, i.e., an instance of one class may not be an instance of another one. Nevertheless, this may be simulated by means of abstract super classes for the equivalent classes. Thus, it is possible to either use objects of the concrete classes or of one of its supertypes. In this respect, equivalent classes are represented by abstract `Class` places. `OWLDataType-Property`, which define attributes of classes, are equivalent to `EAttribute` in Ecore and are mapped to according `Attribute` places. Finally, `OWLObjectProperty` in OWL defines references between classes which are similar to EReference in Ecore and are thus mapped to according `Reference` places. The according individuals are translated to the respective tokens similar to the concepts presented above.

As described before, model transformations are specified between their according metamodels (M2) and are executed on the model level (M1), i.e., every model that conforms to its according metamodel may be handled by the transformation. As can be seen in Fig. 4.10, these metamodels conform themselves to a so-called meta-metamodel (M3, e.g., Ecore). Since Transformation Nets abstract from the concrete metamodels and their models by means of places and tokens, it is possible, e.g., to translate a source metamodel that conforms to the Ecore meta-metamodel into places and the according model to tokens, and to translate a target metamodel that conforms to the OWL metamodel into places and the according individuals into tokens. Consequently, it is not only possible to specify transformations between source and target metamodels (M2) that are specified using the same meta-metamodel (M3, e.g., Ecore), but also between metamodels that themselves conform to different meta-metamodels, as shown in Fig. 4.10.

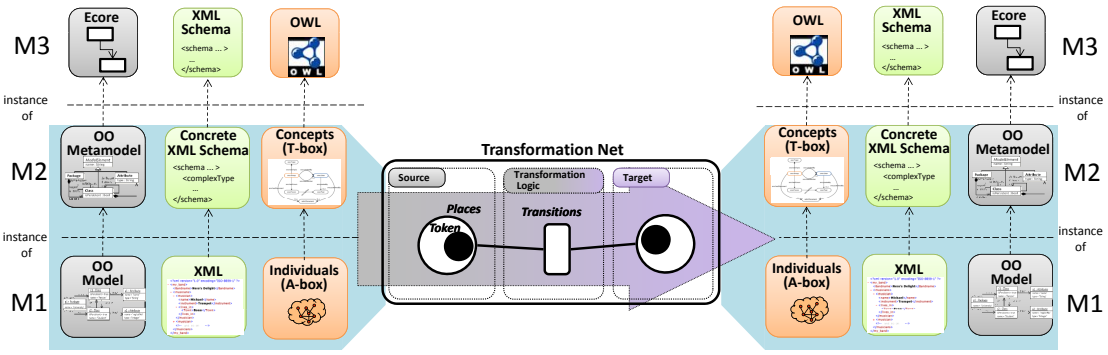


Figure 4.10: Overcoming Meta-Metamodel Heterogeneities in Transformation Nets

Therefore, Transformation Nets not only provide means to overcome structural heterogeneities, i.e., differences resulting from applying different modeling constructs for the same semantic concept in different metamodels that conform to a common meta-metamodel, but also meta-metamodel heterogeneities, i.e., transformation between metamodels that themselves conform to different meta-metamodels are enabled.

4.4 Dynamic Parts of Transformation Nets

The previous section dealt with describing how metamodels and models are represented as the static parts of a Transformation Net. This section introduces the dynamic parts of a Transformation Net. First, the firing behavior of Transformation Nets is explained, followed by a discussion, how conditions and functions may be specified in Transformation Nets. The second part of this section explains, how to chain transitions in Transformation Nets in order to represent complex transformation logic.

4.4.1 Representation of Transformation Logic

An execution of a model-to-model transformation rule has two major phases. The first phase comprises the matching of certain elements of the source model, from which information is derived that is used in the second phase for producing the elements of the output model. This matching and producing of model elements is supported within Transformation Nets by firing transitions whereby the source model is accessed in a read only manner and the target model in a write only manner, i.e., transitions are not allowed to match for elements in the target model. `Transitions` in Transformation Nets are similar to the concept of `TransformationRules`, as described above (cf. Fig. 4.2 in Section 4.2). Transitions are enabled, if a certain configuration of matching tokens is available. This configuration is expressed with the remaining elements of the subpackage *DynamicElement* of the Transformation Net metamodel (cf. Fig. 4.11). To specify the firing behavior of a transition, a mechanism is used that is well known from graph transformation systems [41]. Thereby, two patterns of input and output placeholders for tokens are defined, which represent a precondition and a postcondition (cf. references `Transition.queryPatterns` and `Transition.productionPatterns` in Fig. 4.11). Thereby, a certain configuration of tokens is matched from the input places, and a certain configuration of tokens is produced in the output places. Once such a configuration is found, the transition is enabled and ready to fire. In this respect, these concepts fulfill similar tasks as `InPatterns` and `OutPatterns` in transformation languages. The details of how to specify a transition's firing behavior is described in the following. In order to explicate the concepts, an extract of the running example is used, which translates `Class` instances into according `Table` instances, as shown in Fig. 4.12.

Specification of Transition's Firing Behavior. The firing behavior of transitions in Transformation Nets is defined by means of so-called `Patterns`. As can be seen in Fig. 4.11 the abstract class `Pattern` is refined by the concrete subclasses `ObjectPattern`, `ValuePattern` and `LinkPattern`, which are represented by the same concrete syntax as their according tokens. As the names already imply, different types of patterns are used to either query or produce

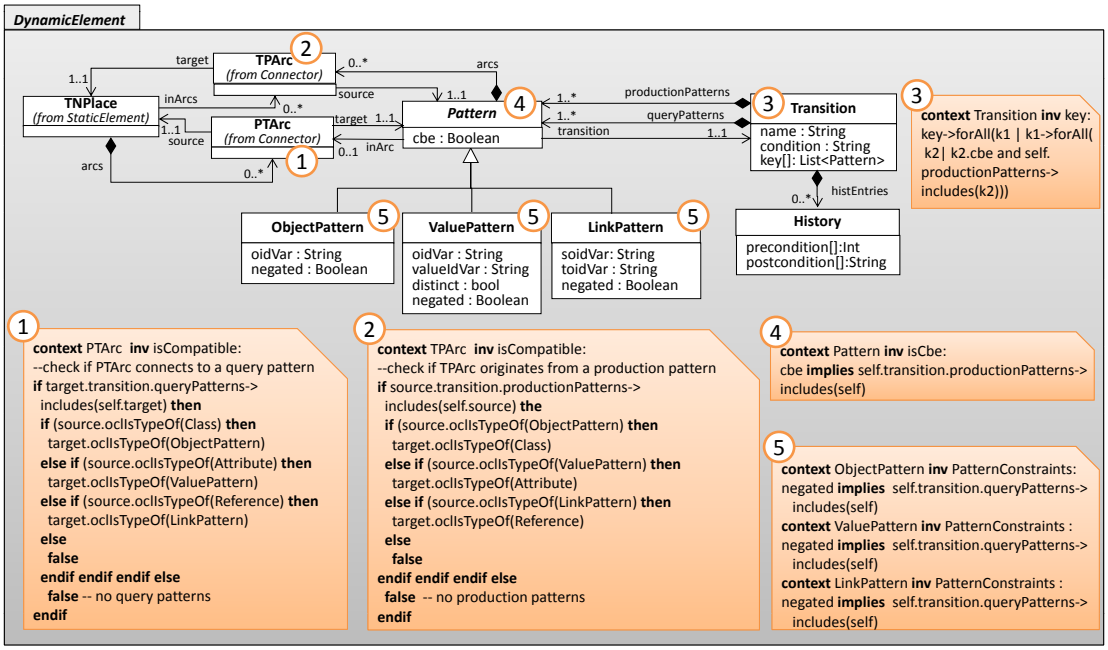


Figure 4.11: Dynamic Elements of Transformation Nets

according types of tokens, e.g., if an object token should be queried, an `ObjectPattern` is used. Patterns may then be used in two different roles, either as `query patterns` to model preconditions or as `production patterns` to model postconditions of a transition, as detailed in the following:

- **Query Patterns**: Query patterns constitute the precondition or left-hand side (LHS) of a transition. Every query pattern has to be connected to a certain place of the source metamodel using an instance of a `PTArc`. It is important that the type of the place (either `Class`, `Attribute`, or `Reference`) corresponds to the according type of pattern, e.g., a `Class` place may only be connected to an `ObjectPattern` (cf. OCL constraints ① and ② in Fig. 4.11). The actual `variables` (represented by the according attributes of the patterns) are bound to the according `ids` of an actual input token during matching. In order to exemplify this, Fig. 4.12(a) depicts a sample transition, containing two query patterns. The pattern labeled with the variable `class` represents an `ObjectPattern` which queries for according objects of the class `Class`. The pattern labeled with the variable `class` and `name` represents a `ValuePattern` which queries the `ModelElement.name` attribute (which may be accessed by classes since `Class` inherits from `ModelElement`). During matching, equal variables have to be bound to equal ids of the tokens, e.g., if the variable `class` of the `ObjectPattern` is bound to token `c1`, then the variable `class` of the `ValuePattern` may only be bound to `c1` as well, i.e., the transition is only enabled, if there exists a `name` attribute for a certain object. Furthermore, if two patterns are connected to the same source place, the according pattern

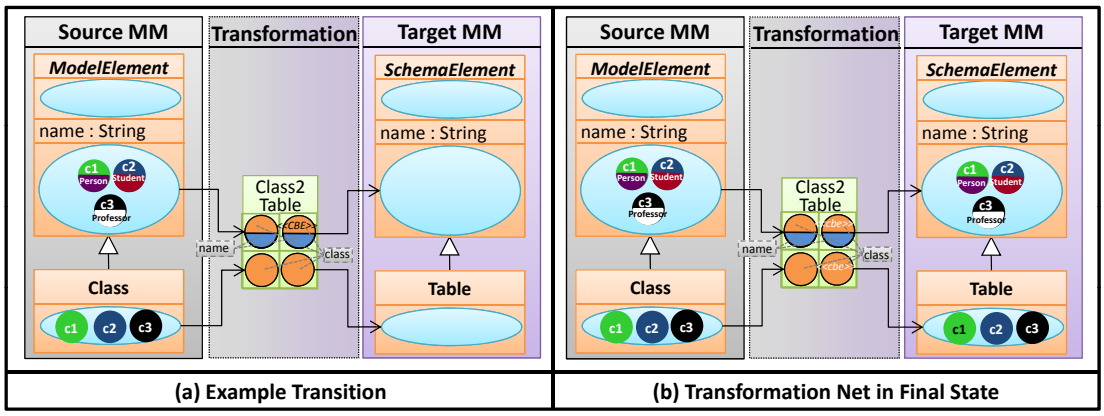


Figure 4.12: Example Transition in Transformation Nets

variables must be different to ensure a non-injective matching, i.e., two objects with the same type in the source pattern cannot get matched to the same object in the model.

Negative Pattern. Besides querying if a certain token configuration is available it is also necessary to query if a certain token configuration is not available, e.g., only if a certain `Class` instance does not exhibit a link `superClass` to a super class (i.e., it is a root class), an according transition should be enabled. Since the matching algorithm of CPNs may only match for the existence of tokens, the handling of non-existence of tokens requires the introduction of list data types and conditions in CPNs (cf. Chapter 6). Since the Transformation Net DSL is intended to hide such complexities from the transformation designer, negative query patterns have been included in the language definition, i.e., query patterns may be negated by setting the `negated` flag to `true` (cf. example 6 in Fig. 4.13)[7]. In this respect, negative patterns are comparable to negative application conditions in graph transformations [61].

Distinct Values. In case of a `ValuePattern`, the situation might occur that only distinct attribute values should be considered, e.g., only `Class` instances offering a different value of the `Class.name` attribute should be translated into according `Table` instances. Again this situation requires the handling of list data types and conditions in CPNs and should therefore be hidden from the transformation designer in the Transformation Net DSL. Thus, a distinct flag on `ValuePatterns` is provided to ease the specification (cf. `ValuePattern.distinct` in Fig. 4.11 and in Fig. 4.13).

- **Production Patterns**: Production patterns constitute the postcondition or right-hand side (RHS) of a transition. Every production pattern has to be connected to a place of the target metamodel by means of `TPArc` instances. The variables of `ProductionPatterns` depend on the variables of the `QueryPatterns`. In the example in Fig. 4.12(a), both `ObjectPatterns` (query as well as production patterns) exhibit the variable `class`.

[7] The flag is contained in the subclasses since later on the metamodel will by extended by another pattern, which is not allowed to be negated.

This means if, e.g., the object token `c1` is bound to the variable `class` of the query pattern during the matching phase, the variable `class` of the production token has to be bound to `c1` when firing the transition. In this respect, the object is simply copied from a source place to a target place, as can be seen in Fig. 4.12 (b), which shows the Transformation Net in its final stage, i.e., all transitions have fired. However, it should also be possible to produce new objects, values or links in the target model which do not exist in the source model, e.g., if several source objects are merged to a new target object. In this case, a newly colored token should be produced, in order to accordingly visualize the new element. In Transformation Nets this may be achieved if a certain variable of a production pattern has not been used as a variable of a query pattern.

Check Before Enforce. Similar to distinct value query patterns, mechanisms for production tokens are needed to ensure that duplicate elements are not created when the required elements already exist, i.e., if a `Schema` has already been created for a certain `Package`, the schema should be reused and no duplicates should be produced. For this production patterns may be marked as check before enforce (cf. `Pattern.cbe` in Fig. 4.11 and example 2 in in Fig. 4.13). As may be seen in the example in Fig. 4.12, the production `ValuePattern`, which produces the `name` attribute and its corresponding containing `ObjectPatten` are marked as check before enforce, i.e., only differently named `Table` instances will result. In order to provide flexibility in specifying equality, the transformation designer is allowed to define a key by specifying the according production patterns, which have to marked as check before enforce.

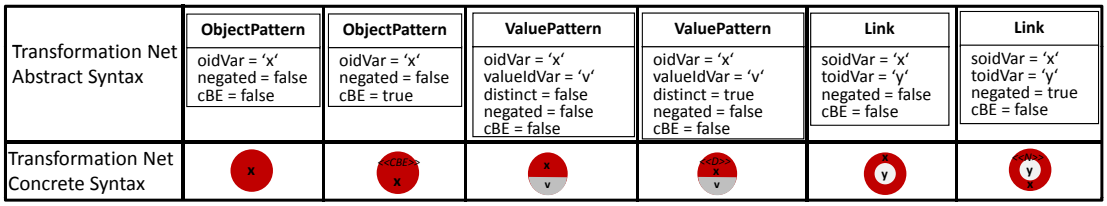


Figure 4.13: Overview on Concrete Syntax of Patterns in Transformation Nets

Firing Behavior. Transformation Nets exhibit a different default firing behavior than standard CPNs in the sense that transitions in Transformation Nets do not consume tokens per default. This is since, on the one hand, all possible token combinations must be taken into account. For example, if a transition matched `Package` tokens and `Class` tokens at once, the transition could fire only once, although if multiple elements were available, since there is a 1:n relationship between `Package` and `Class`. On the other hand, if more than one transition accessed a certain place, consuming firing behavior would lead to erroneous race conditions. By default, every transition is just reading the tokens of the connected input places and does not delete them. In order to prevent a transition to fire more than once for a certain token configuration, the already processed configurations are stored in a *history* (cf. reference `Transition.histEntries` and Class `History` in Fig. 4.11). Every `History` entry

stores the `ids` of the matched tokens in the attribute `History.precondition` and the produced tokens in the attribute `History.postcondition`. A transition is only allowed to fire if the current configuration is not found in the history entries' preconditions. Besides prohibiting multiple firings of transitions, the history thus, also reveals trace information, i.e., an explicit correlation of which input tokens have been used to create certain output tokens.

4.4.2 Conditions and Functions

Conditions, e.g., to select a subset of potential source objects, values or links as well as functions, e.g., to calculate and derive values, are key to any model transformation language. For the specification of such conditions and functions most of today's transformation languages employ OCL [115]. Therefore, Transformation Nets make use of OCL as well to allow the transformation designer to specify conditions and functions.

Conditions. The running example demands that only persistent `Class` instances are transformed into according `Table` instances. In this respect, the Transformation Net depicted in Fig. 4.14(a) exhibits an according OCL condition. Since conditions influence the firing behavior of transitions, i.e., a transition is only allowed to fire if the according condition is fulfilled, conditions in Transformation Nets are specified on transitions (cf. `Transition.condition` in Fig. 4.11). In order to actually evaluate OCL expressions, a so-called *OCL context* is needed which represents the root of the OCL expression and thus, the starting point of the evaluation.

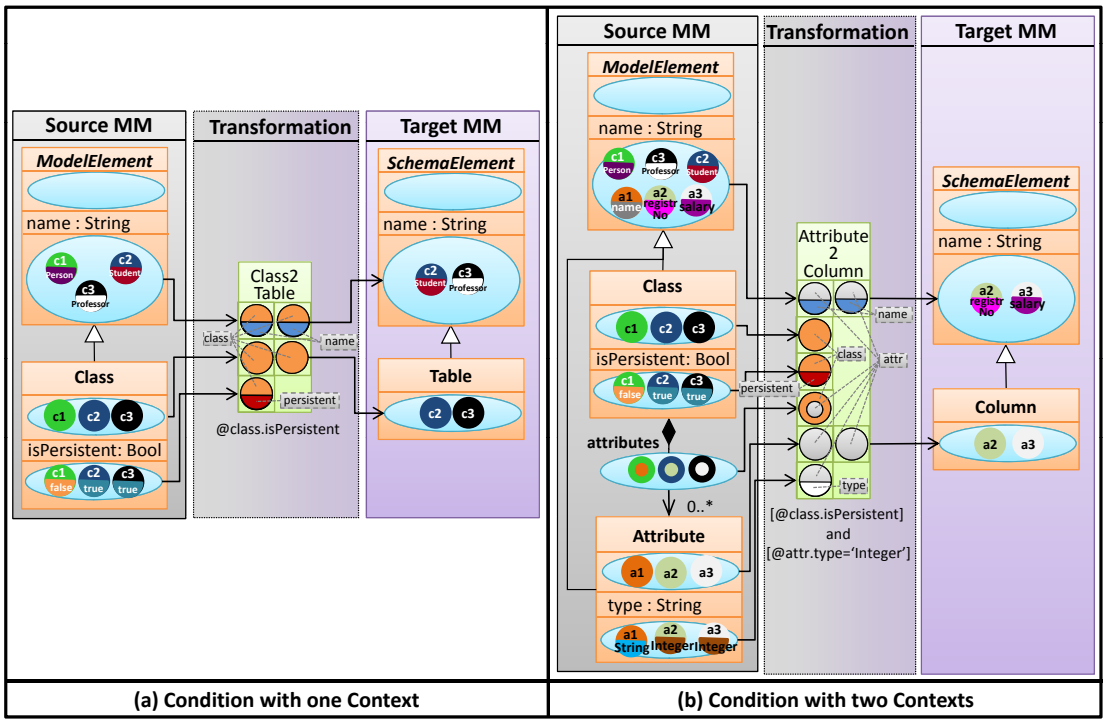


Figure 4.14: Example Conditions

The context of an OCL expression in Transformation Nets is specified by using a variable of a query pattern. In order to be able to distinguish variables from the remaining OCL expression strings, the according variable is preceded by an "@" sign. To specify the condition of the example, the context is set to the according class object which is queried by means of the object pattern exhibiting the variable `class`, i.e., the context is set by means of `@class` in the expression. Since the OCL context is determined by an object and it's according class, only variables of query patterns typed to `ObjectPattern` are valid. As soon as the context is determined, any valid OCL expression may be specified. In this respect, the `isPersistent` attribute of the class `Class` may be queried. Therefore, the complete condition of the example is `@class.isPersistent`.

Unfortunately, OCL allows for one context per expression only which makes it cumbersome to specify more complex OCL constraints and requires potentially complex navigation expressions. For example, the specification of the condition "if the class is persistent and if the type of an according attribute is integer" would require various navigation expressions. To ease the specification and to provide flexible means to express and evaluate conditions, several contexts are allowed in Transformation Nets. Nevertheless, in order to allow for the reuse of existing OCL engines, it is required to explicitly delimitate the scope of a context. Brackets are used for this purpose in Transformation Nets as can be seen in Fig. 4.14(b). To evaluate these conditions, the OCL expression is split along the different contexts and then every context is evaluated separately. Afterwards, the separate results are evaluated according to the boolean operators that connected the individual contexts. Concerning the example, first the according binding of `class.isPersistent` is checked and second the binding of `attr.type`. Only if both parts of the condition are fulfilled, the whole condition holds. Therefore, only the bindings `class=c2`, `attr=a2`, `persistent=true`, `name=registrNo`, `type=Integer` and `class=c3`, `attr=a3`, `persistent=true`, `name=salary`, `type=String` enable the transition.

Functions. As already mentioned, OCL may not only be employed to specify conditions but also for functions, e.g., to concatenate values. Since a single production pattern might exhibit several outgoing arcs, functions are specified on the according `TPArcs`. The specified function may depend on the query tokens, thus the variables in the function have to be variables of the query patterns. The example shown in Fig. 4.15 concatenates the name of a class with the postfix `_gen` by adding the function `@class.name.concat('_gen')` to the outgoing arc. Since in this case a new value is generated the according production attribute pattern exhibits a new variable `newName`, i.e., the variable `newName` has not already been used by a query token. Therefore the color representing the vale of the attribute `SchemaElement.name` exhibit a color, that does not exist in tokens of the source model.

4.4.3 Chaining of Transitions

The presented transformations so far solely consist of a single transition only and match tokens of the source model and directly produce tokens in the target model which would correspond to a single rule in current transformation languages. Nevertheless, a transformation usually consists of several rules which have to interact with each other, i.e., a chaining of transformation rules is required. Thus, Transformation Nets provide two different means to chain the according transi-

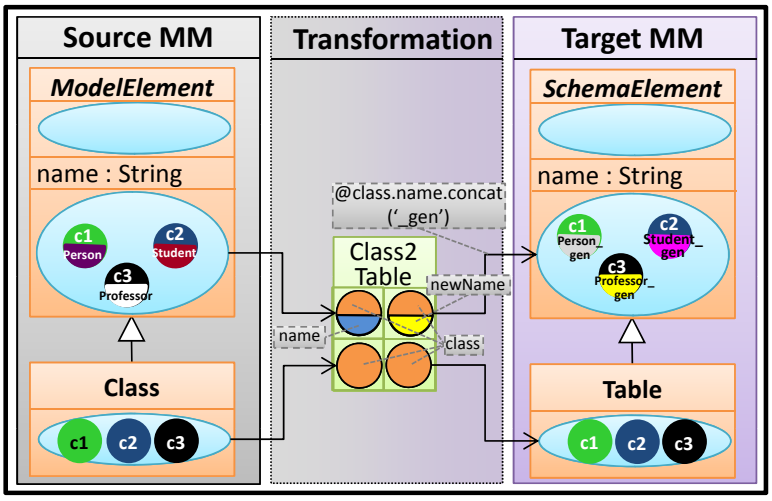


Figure 4.15: Example Function

tions, being (i) *trace information* and (ii) *intermediate places*. On the one hand, trace information may be used to define dependencies between transitions in a way that a transition may only fire if another transformation has already produced some elements, i.e., trace information. On the other hand, intermediate places are places that are neither part of the source metamodel nor the target metamodel; instead these places may be used to make derived information explicit for further transformations. An example thereof is the calculation of the transitive closure, since typically only the direct superclasses are stored in the model (cf. reference `Class.superclasses` in the running example), but not its indirect superclasses, e.g., in the context of our running example (cf. Fig. 1.2 on page 5) the indirect inheritance relationship between the class `Professor` and `Person` should be made explicit.

Trace information. Two requirements of the running example are to (i) create an according `Table` instance for every persistent `Class` instance and (ii) to create according `Column` instances for direct attributes of such classes in a first step (cf. requirements 2 and 3 in Section 1.2). When inspecting the source metamodel, one may detect that `Classes` and `Attributes` are linked by the unbounded reference `Class.attributes`. If the transformation designer specified a single transition that matches for `Class` and `Attribute` instances that produced according `Table` and `Column` instances, this would result in potentially too many `Tables` (without using check before enforce production patterns). This is since there are too many matches in case a class exhibits more than one attribute. To avoid this, the transformation designer may specify two transitions, whereby the transition that generates `Column` instances for `Attribute` instances should only be enabled if the according `Class` instances have already been transformed to `Table` instances. To achieve this, *trace information* may be applied in Transformation Nets. Trace information makes explicit which source object(s) have been translated into which target object. In this respect, the parts of the history that concern objects are made explicit. The remaining parts of the history are not made explicit since dependent transi-

tions need to know the according object only, e.g., if a dependent transition sets a link, it only needs to know the according source and target objects. To incorporate trace information in Transformation Nets, in a first step, the `StaticElement` and `DynamicElement` packages are extended by according metamodel elements, i.e., `TracePlace`, `Trace` and `TracePattern` (cf. highlighted elements in Fig. 4.16(a)). `Trace` tokens and `TracePatterns` exhibit both two attributes to store the information which source elements – multivalued attributes `Trace.soid` and `TracePattern.soidVar`, left side of token or pattern in concrete syntax – have been translated to which target element – `Trace.toid` and `TracePattern.toidVar`, right side of token or pattern (cf. Fig. 4.16(b)).

Fig. 4.17 shows the application of trace information to solve the above stated requirements of the running example. The transition `Class2Table` provides trace information produced

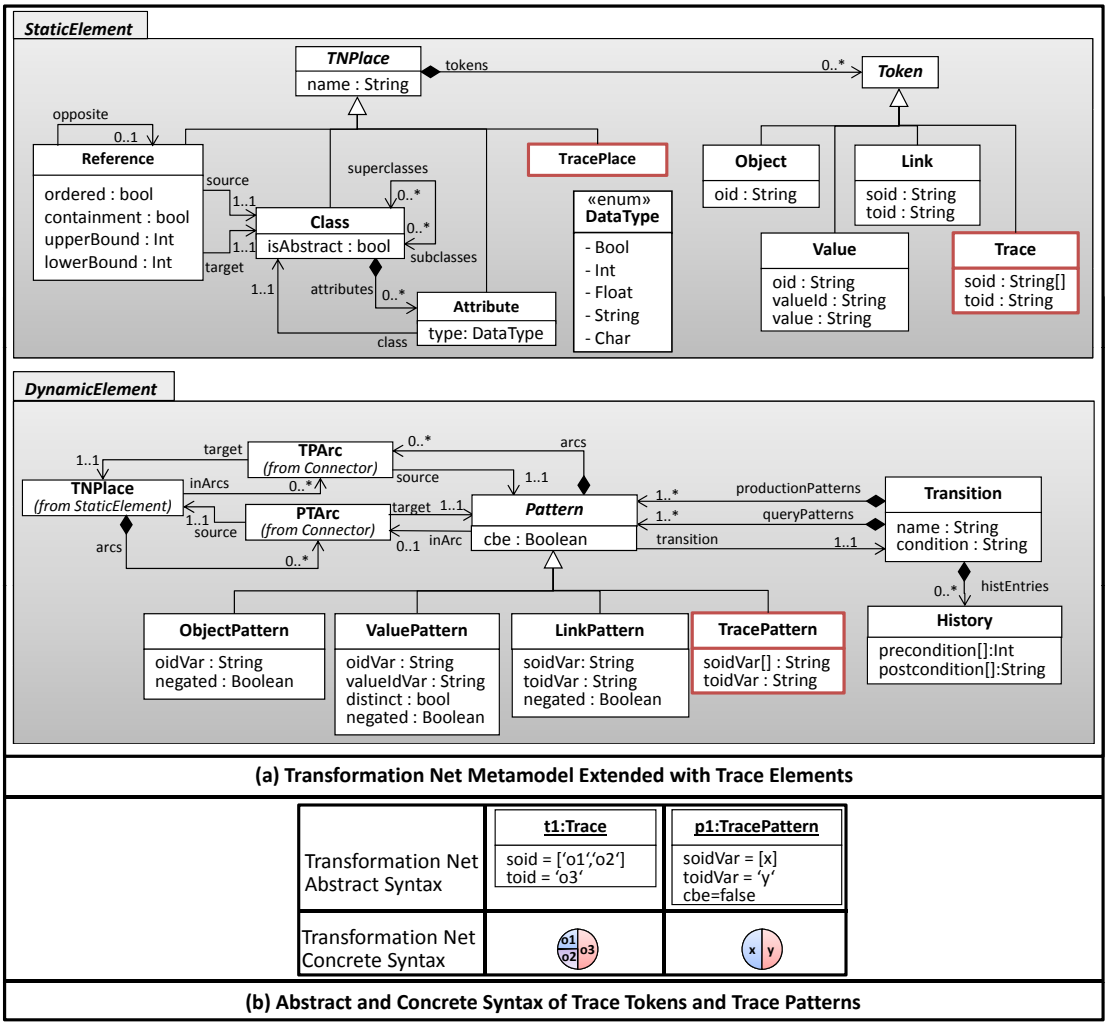


Figure 4.16: Extension of Transformation Net Metamodel to Represent Trace Information

by the production `TracePattern` with the two variables labeled to `class`. The produced `Trace` tokens are stored in the `ClassTrace` place (cf. place with stereotype `TracePlace` in Fig. 4.17). Since in this example, `Class` objects are only copied to `Table` objects, the trace tokens are of one color only. The produced trace information may then be queried by subsequent transitions. In the example, an `Attribute` instance should only be translated to a `Column` instance if the according `Class` instance has also been translated to a `Table` instance. The transition `Attribute2Column` does not query the `Class` source place but uses the *ClassTrace* place to match for `Class` instances that have already been translated to `Table` instances. As can be seen in Fig. 4.17, the transition `Attribute2Column` uses a `TracePattern` to query the according trace information. Since in general the generated target object needs not to exhibit the same color (i.e., a new object could have been created), the `tracePattern` must be capable to handle this situation as well. Therefore, the query `TracePattern` exhibits different variables (colors) on its left (source) and right (target) side. Nevertheless, during matching it is allowed that two different variables are bound to the same `id` (color), e.g., in the example both, variable `class` and `table` may be bound to the same `id`. The transition `Attribute2Column` is thus only enabled if a `Table` instance has already been created and if a certain `Attribute` (`attr` query token) is contained in the according `Class`, expressed by a reference query token labeled to `class` and `attr`. In this respect, the transitions `Class2Table` and `Attribute2Column` interact with each other, since the transition `Attribute2Column` queries the required trace information from the transition `Class2Table`.

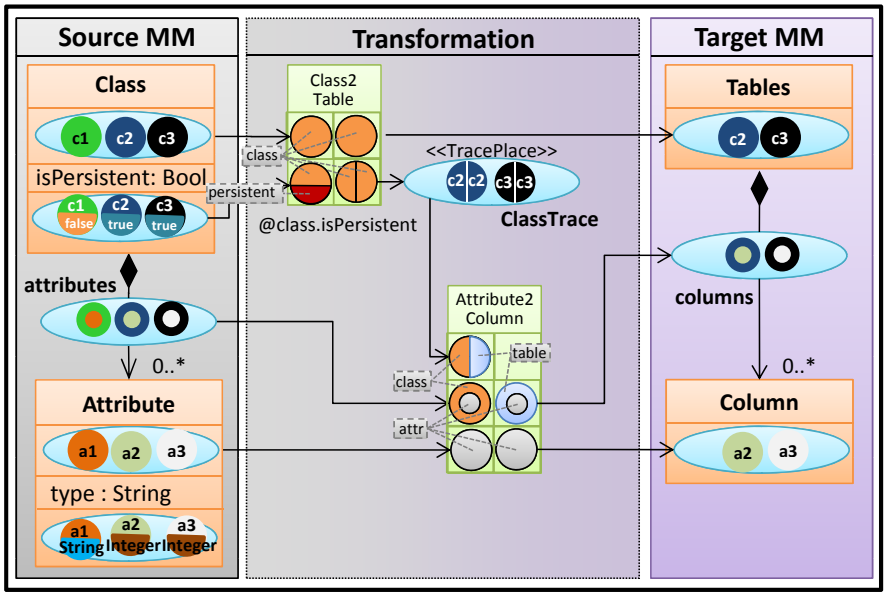


Figure 4.17: Example Transition using Trace Information

Intermediate Places. The last requirement of the running example is that during the transformation process, for each directly or indirectly inherited `Attribute` instance, a correspond-

ing `Column` instance should be generated. Nevertheless, the specified metamodel provides means to access direct superclasses by means of the reference `Class.superClasses` only (cf. Fig. 1.2). Therefore, first indirect subclasses need to be derived, i.e., the transitive closure has to be calculated, in order to be able to generate according `Column` instances. To make such derived information explicit, so-called *intermediate places* may be used. In this respect, the example depicted in Fig. 4.18 extends the previous example by using an intermediate reference place (cf. place `Closure` in Fig. 4.18) to store the calculated transitive closure to be further on able to generate `Column` instances for indirectly inherited `Attribute` instances. To calculate the transitive closure, first the `Helper` transition copies the links to superclasses into the intermediate place `Closure`. The transition `TransitiveClosure` then reads these tokens and calculates the transitive links, i.e., it checks if there exists a link from a `subClass` to a `superClass` and from the `superClass` to another super class, denoted by the variable `baseClass`. In the example in Fig. 4.18 one additional link from class `c3` to `c1` is generated and put into the intermediate place `Closure`. This is since the class `c3` inher-

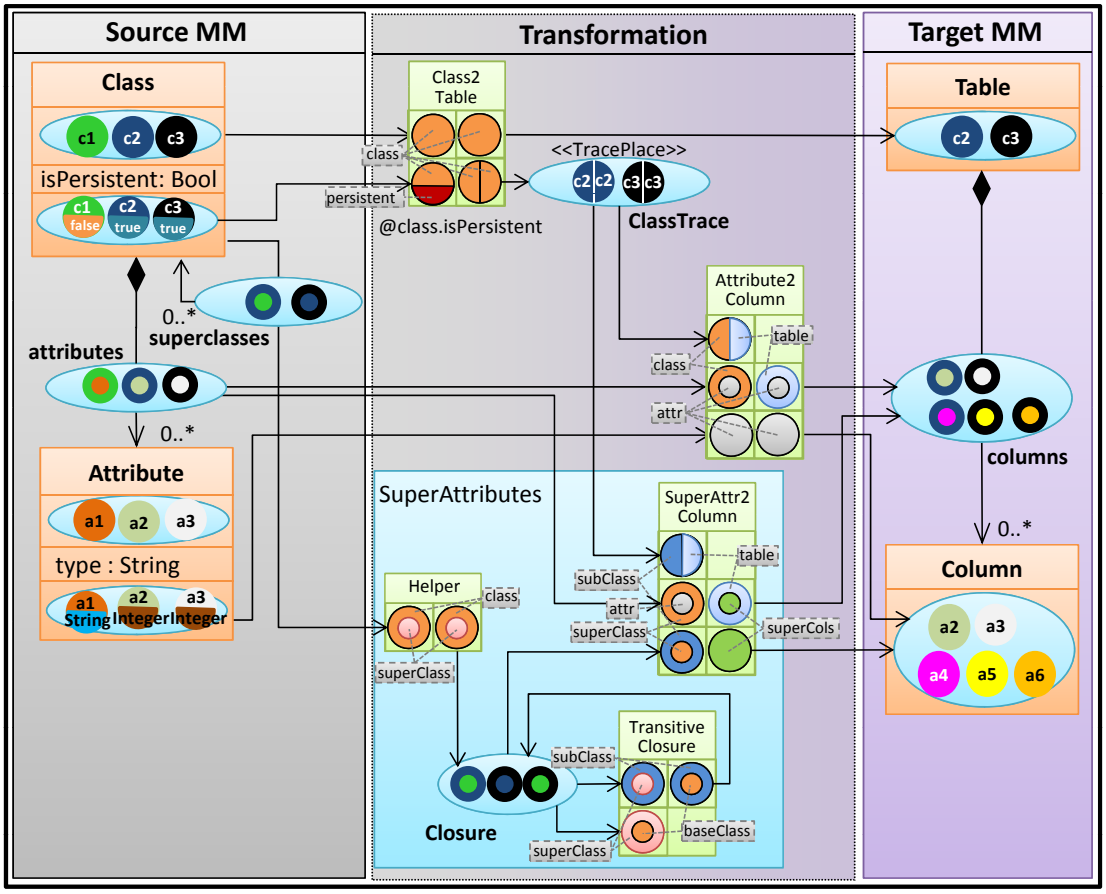


Figure 4.18: Example Transition using Intermediate Places

its from c2, which inherits again from c1 as may be seen from the tokens in the reference place superClasses. The tokens in the intermediate place are then used by the transition SuperAttribute2Column to produce the remaining Column objects and columns links. Thereby, the transition uses the trace information provided by the transition Class2Table in order to add the Column instances that need to be generated to the corresponding table. The last query token of the transition SuperAttributeToColumn queries the transitive links, i.e., if there is a certain link from a subclass to a superclass, i.e., it is searched for indirect subclasses. If this superclass has links to according attributes, the transition is ready to fire and produces a column for every inherited attribute as well as the according links.

To exemplify this, in case of class c3, which inherits directly from c2 and indirectly from c1, two additional columns (a5 and a6) are created. Since the context token stemming from the source class c3 may be bound to the context query token and since there are links from c3 to c2 and from c3 to c1 (cf. tokens in intermediate place) as well as references from c2 and c1 to according attributes available, the transition may fire twice to produce the columns a5 and a6 as well as the according links. In this way, for every directly and indirectly inherited Attribute instance a corresponding Column instance is generated.

4.5 Modularization Concepts in Transformation Nets

Whereas the previous sections introduced the basic concepts of Transformation Nets, this section presents *modules* as a modularization concept in Transformation Nets. In this respect, modules allow the transformation designer to divide a transformation specification into self-contained, maintainable parts, which are discussed in the following.

4.5.1 Overview on Modules

Modules encapsulate a certain transformation logic and provide a well defined interface to its environment. In this respect, the metamodel depicted in Fig. 4.19 highlights the elements specific to modules. It may be seen that a transformation specification might contain Modules which may be nested again, i.e., a module may contain several other, more fine-grained modules (cf. reference Module.children). The interfaces are defined by means of Ports of different types, comprising class, attribute, reference and trace ports. Ports are connected with Places, Patterns or Ports of nested modules via Arcs. The internals of modules consist of Transitions and Places, i.e., either trace places or intermediate places.

4.5.2 Two Views on Modules

Since modules are used to modularize the transformation specification, they should also hide unnecessary details in order to make the approach scalable for large transformations. In this respect, two different views on modules are provided, being (i) a so-called *blackbox view*, which only shows the interfaces of a module and (ii) a so-called *whitebox view*, which shows the details of the implementation of the modules in Transformation Nets. In the following, the different views are discussed in more detail.

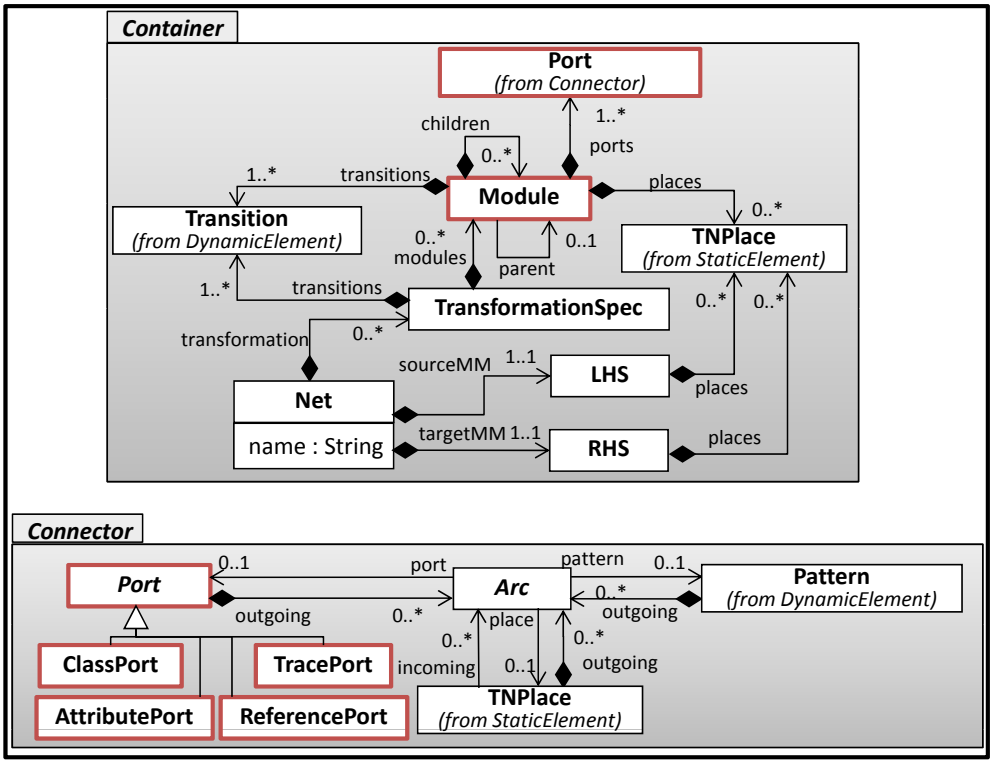


Figure 4.19: Extension of Transformation Net Metamodel to Represent Modules

Blackbox View. The blackbox view on modules exhibits only its interface and the actual bindings to a metamodel. As can be seen in Fig. 4.20, the example depicts two modules, where the module `Package2Schema` translates `Package` instances to `Schema` instances and the `Class2Table` module translates `Class` instances to `Table` instances, encapsulating requirement one and two of the running example. In this example only class and trace ports are shown in the blackbox view. This is since classes set the focus of the transformation and trace ports make the dependencies between modules explicit. Transformations of corresponding attributes or references are then specified in the whitebox view, as discussed in the next paragraph. In this respect, the interface of a module should be minimal, i.e., only a minimum number of ports should be exhibited in the blackbox view. For example, although it would be possible to attach an attribute port to both modules in Fig. 4.20, this is omitted since such details should rather be specified in the whitebox view, as discussed below.

Whitebox View. The actual implementation of the transformation logic encapsulated in a module is shown in the whitebox view. The whitebox view of a module may contain an arbitrary number of transitions as well as trace and intermediate places, i.e., arbitrary complex transformation logic may be encapsulated. Fig. 4.21 shows the whitebox view of the modules `Package2Schema` and `Class2Table`. In this example, first `Package` instances and their according `name` attributes are copied to `Schema` instances, which are equally named (cf. fired

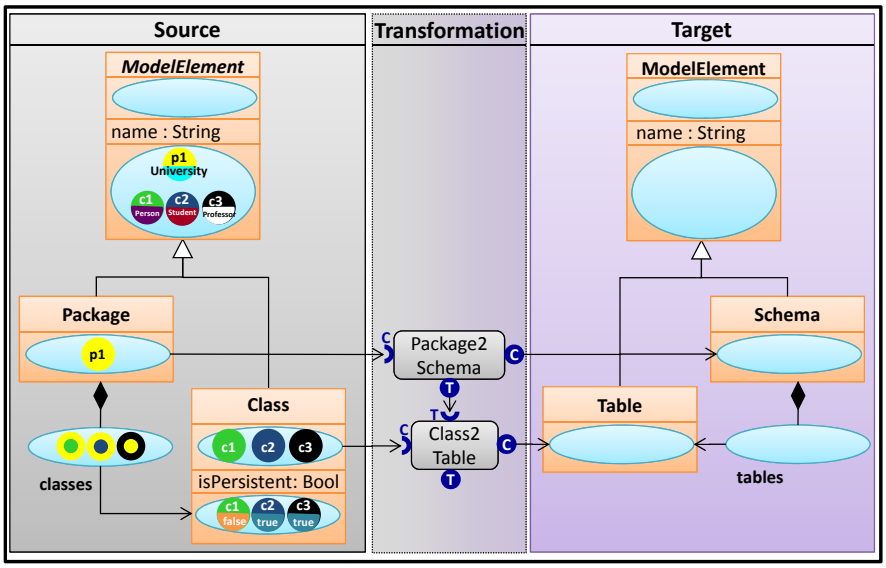


Figure 4.20: Blackbox View on Modules

transition in Fig. 4.21(a)). Additionally, the transition produces trace information in order to provide the information, which `Package` instance has been translated to which `Schema` instance for dependent transitions which may be contained in different modules. This trace information is then used by the module `Class2Table` which queries for persistent, named `Class` instances and copies them to equally named `Table` instances (cf. transition in Fig. 4.21(b)). The trace information is used to create an according link to the `Schema` instance, the generated `Table` instance should be contained in.

Besides the transformation logic itself, the whitebox view restricts the source and target-metamodel to the extracts that were bound in the blackbox view. Therefore, e.g., the whitebox view of the module `Package2Schema` shows only the metamodel elements `Package` and its superclass `ModelElement`. The transformation designer is then only allowed to use these elements in the transformation logic, e.g., only the `name` attribute may be additionally used.

Modules may not only be used to modularize Transformation Nets, but they may also be applied to implement patterns, i.e., patterns to resolve recurring structural heterogeneities as done in [86]. Since the ports of modules are only typed to classes, attributes, references or trace information, transformation logic may be implemented without relying on the types of a specific metamodel. This encapsulated transformation logic may then be reused in other transformations by simply binding the ports to the according metamodel elements.

4.6 Summary

In summary, this section presented Transformation Nets as a runtime model for model-to-model transformations. Transformation Nets build a DSL on top of CPNs tailored to the domain of model transformations, hiding the actual details and complexity of CPNs. Metamodel

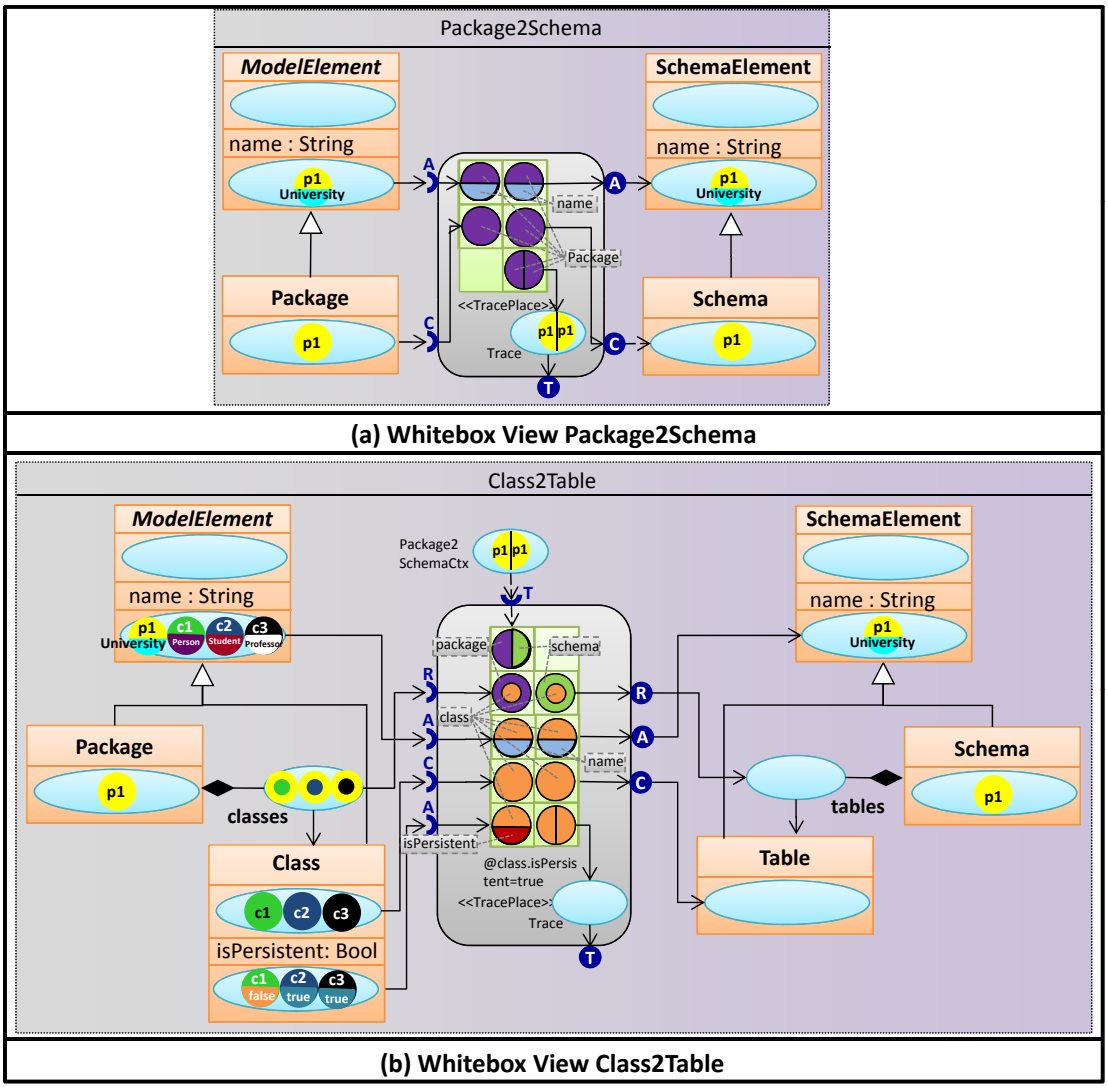


Figure 4.21: Whitebox View on Modules

elements are represented as places and models are made explicit by means of tokens, residing in the corresponding places. The actual transformation logic is specified using transitions which match for tokens residing in places, representing the source metamodel and produce tokens which are put into the places representing the target metamodel. Finally, modules as a means for modularization of the transformation specification were introduced.

In comparison to [125], where an initial version of Transformation Nets has been proposed, numerous improvements and extension have taken place, especially adaptations in the underlying metamodels. In [125], only two types of tokens and patterns (`one-colored` for objects and `two-colored` for attribute and references) have been distinguished. This leads to the fact

that in the transformation logic it was impossible to distinguish between attributes and references due to missing type information, which led to error-prone specifications. In order to ensure correct typing, three explicit types (`Class`, `Attribute`, `Reference`) have been introduced in the course of this thesis to ensure typing to the common core-concepts of metamodels. Furthermore, the visual concrete syntax has been adapted in order to minimize the gap between class diagrams, which are the prevalent notation for metamodels in MDE, and Petri Nets. As a further new contribution, it has been shown that not only object-oriented metamodels and models may be compiled to Transformation Nets, but also XML schemas and ontologies as well as their corresponding instances. Considering the dynamic aspects of Transformation Nets, a major improvement imposed the inclusion of OCL conditions and functions to transitions. Furthermore, the concept of trace places and trace tokens was introduced in order to allow for chaining of transitions, i.e., one transition may query the elements that were produced by another transition. In the following Chapter 5, concepts of Transformation Nets will be introduced that deal with inheritance between transitions in order to cope with rule inheritance in transformation languages. Finally, the version presented in [125] did not exploit the power of CPNs, since Transformation Nets were not translated to standard CPNs for execution and state space analysis, but only made use of a proprietary execution engine developed in Java. The translation of Transformation Nets to standard CPNs will be presented in the subsequent Chapter 6.

Chapter 5

Rule Inheritance in Transformation Nets

We are what we repeatedly do.
Excellence then, is not an act, but a habit.

— *Aristotle*

Contents

After the previous section introduced the fundamental concepts of Transformation Nets, this section focuses on rule inheritance in model-to-model transformation languages and their according representation in Transformation Nets. Since existing model transformation languages exhibit different semantics of rule inheritance concepts, a detailed comparison is pursued first. Three different comparison dimensions are considered, being (i) *syntactic aspects*, i.e., which language constructs are needed to express inheritance between transformation rules, (ii) *static semantics*, i.e., whether a set of inheriting transformation rules is well-formed at compile-time and (iii) *dynamic semantics*, i.e., how inheriting rules interact at run-time.

5.1 Rule Inheritance in Current Transformation Languages

The previous chapter dealt with common concepts of all transformation languages, e.g., every model-to-model transformation language has to provide means to query for source elements and to produce target elements. In order to support large transformation scenarios reuse mechanisms are indispensable. Although the concept of inheritance plays a major role in metamodels (MMs) (as revealed, e.g., by the evolution of the UML standard [100]), inheritance between transformation rules has received little attention so far [79]. Currently only a few declarative, rule based model-to-model transformation languages allow to inherit between transformation rules. As inheritance is employed in MMs to reuse feature definitions from previously defined classes, inheritance between transformation rules is indispensable in order to avoid code duplication and consequently maintenance problems in a transformation specification. The situation is further aggravated by the fact that the provided language constructs to specify rule inheritance and the inheritance semantics differ considerably between the transformation languages. Therefore, first issues in rule inheritance are identified and second, a comparison of the inheritance mechanisms provided by the declarative model-to-model transformation languages supporting rule inheritance is conducted.

5.1.1 Issues in Rule Inheritance

To compare rule inheritance in transformation languages, one starting point is to look at the well-known model transformation pattern (cf. Fig. 5.1) and to examine where the introduction of inheritance would play a role. Obviously, a transformation language must define syntactic concepts (cf. question 1 in Fig. 5.1), which leads to the first dimension of the comparison, namely *syntax*. In this respect, the following questions are of interest:

- *Types of inheritance:* Does the transformation language support only single or multiple inheritance?
- *Abstract rules:* Is it possible to specify transformation behavior that is purely inherited but may not be executed on its own?

In addition to syntax, further well-formedness constraints on the transformation rules must hold (cf. question 2 in Fig. 5.1), which represents the second dimension, namely *static semantics*. Thereby, the following questions may arise:

- *Modification possibilities in subrules:* How may the types and number of input and output elements be changed in subrules such that they may be interpreted in a meaningful way?
- *Unambiguousness issues:* Are there sets of rule definitions that do not allow selecting a single rule?

If a declarative transformation specification is well-formed, it may be compiled into executable code, which is interpreted by a transformation engine that takes a source model and tries

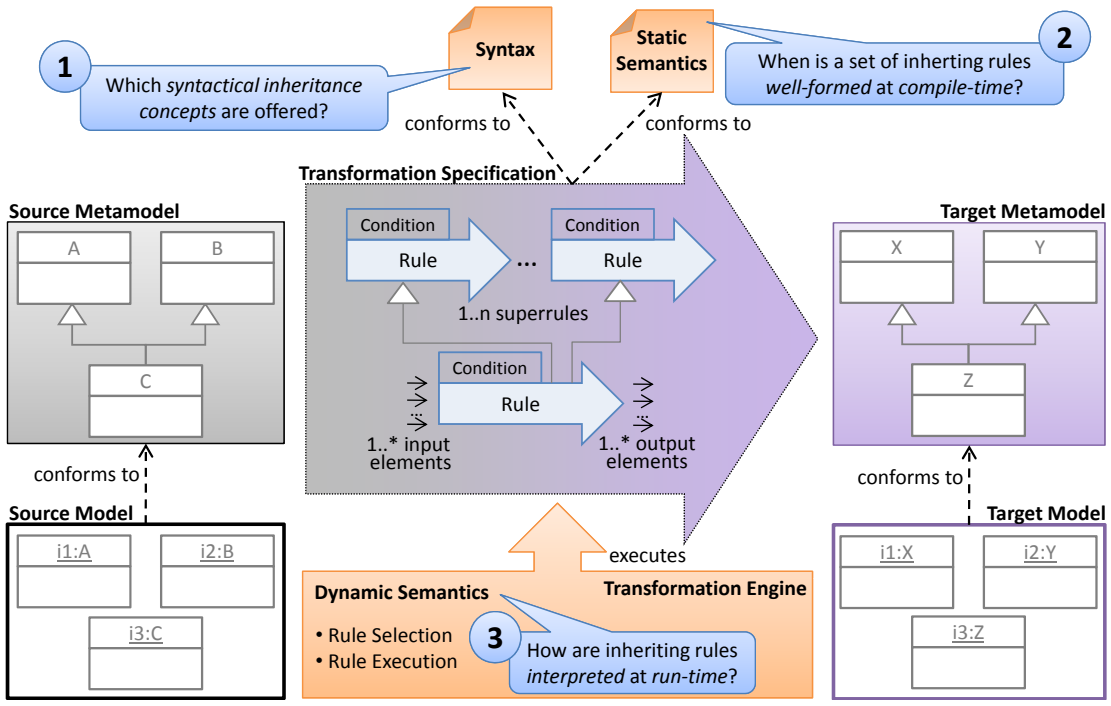


Figure 5.1: Issues in Rule Inheritance

to select and execute rules in order to generate a target model. Again, several questions concerning the interpretation of inheritance at run-time arise (cf. question 3 in Fig. 5.1), which leads to the third dimension, namely *dynamic semantics*:

- *Scope of inheriting rules:* If a rule is defined for a supertype, are the instances of the subtype also affected by this rule?
- *Execution order of inheriting rules:* Are inheriting rules executed top down or bottom up in the rule inheritance hierarchy?

As shown in Fig. 5.2, the criteria may be divided into the three dimensions of (i) *syntax*, (ii) *static semantics*, and (iii) *dynamic semantics* [80]. These dimensions and the corresponding criteria are described in the following, whereby first existing approaches are compared and then in each case the according realization in Transformation Nets is discussed.

5.1.2 Comparison Setup

The comparison of inheritance support in model-to-model transformation languages is based on a carefully developed test set, which includes at least one test case for each criterion, which are presented in detail in the following (cf. Fig. 5.3). These documented test cases, including the example code, the metamodels, and source models, may be downloaded from the project

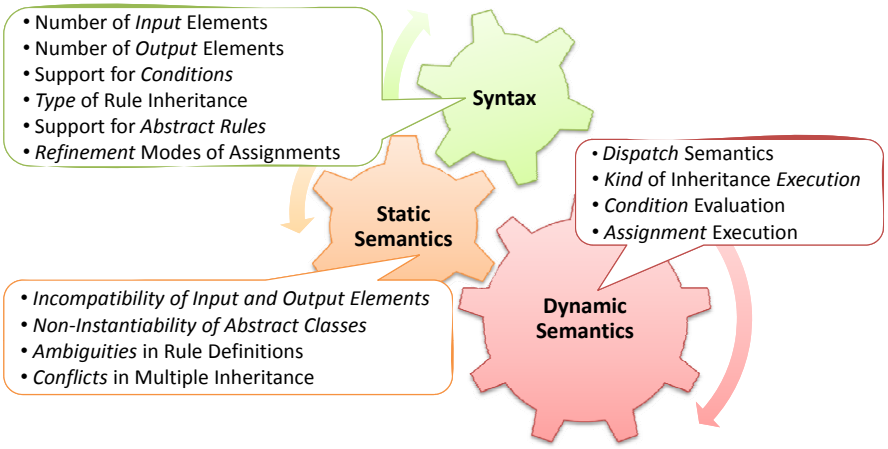


Figure 5.2: Overview on the Comparison Framework

homepage[1]. For the comparison, common model-to-model transformation languages are considered which offer dedicated inheritance support and allow relationships between source and target models to be specified in a declarative way. In this respect, only the declarative subsets of the hybrid transformation languages ATL (version 3.1.0) and ETL (version 0.9.0) as well TGGs provide rule inheritance support. Concerning TGGs, none of the different available implementations actually implements rule inheritance. Nevertheless, TGGs were included in the comparison, since specific literature concerning inheritance support exists [79]. The examples presented based on the MOFLON execution engine of TGGs (MOFLON 1.5.1) In order to be able to compare the bidirectional TGG-based model transformation approach with the unidirectional languages ATL and ETL, only the unidirectional forward translation is considered in TGGs. Please note that although the QVT standard specifies the declarative transformation language QVT Relations, it is not included in this comparison, since QVT Relations support only redefinition of whole rules (without being able to reuse original rule definitions) and no inheritance between rules. Actual refinement of transformation rules is only mentioned in the QVT Core part, which leaves the transfer to QVT Relations open.

Fig. 5.3 shows a potential starting point for the implementation of the running example making use of rule inheritance in ATL, ETL and TGGs. For testing purposes it should be possible to instantiate every target class – therefore the class `SchemaElement` was changed from an abstract to a concrete class. The rule `Class2Table` transforms persistent `Class` instances into corresponding `Table` instances, while inheriting from the rule `MElement2MElement`, which specifies the `name` assignment. In the context of transformation rules, both feature assignments and conditions should be inheritable to subrules. Basically this means when executing the subrule `Class2Table`, the assignments of the superrule `MElement2SElement` should be executed as well. The following sections refer to this example to clarify the details thereof.

[1] http://www.modeltransformation.net

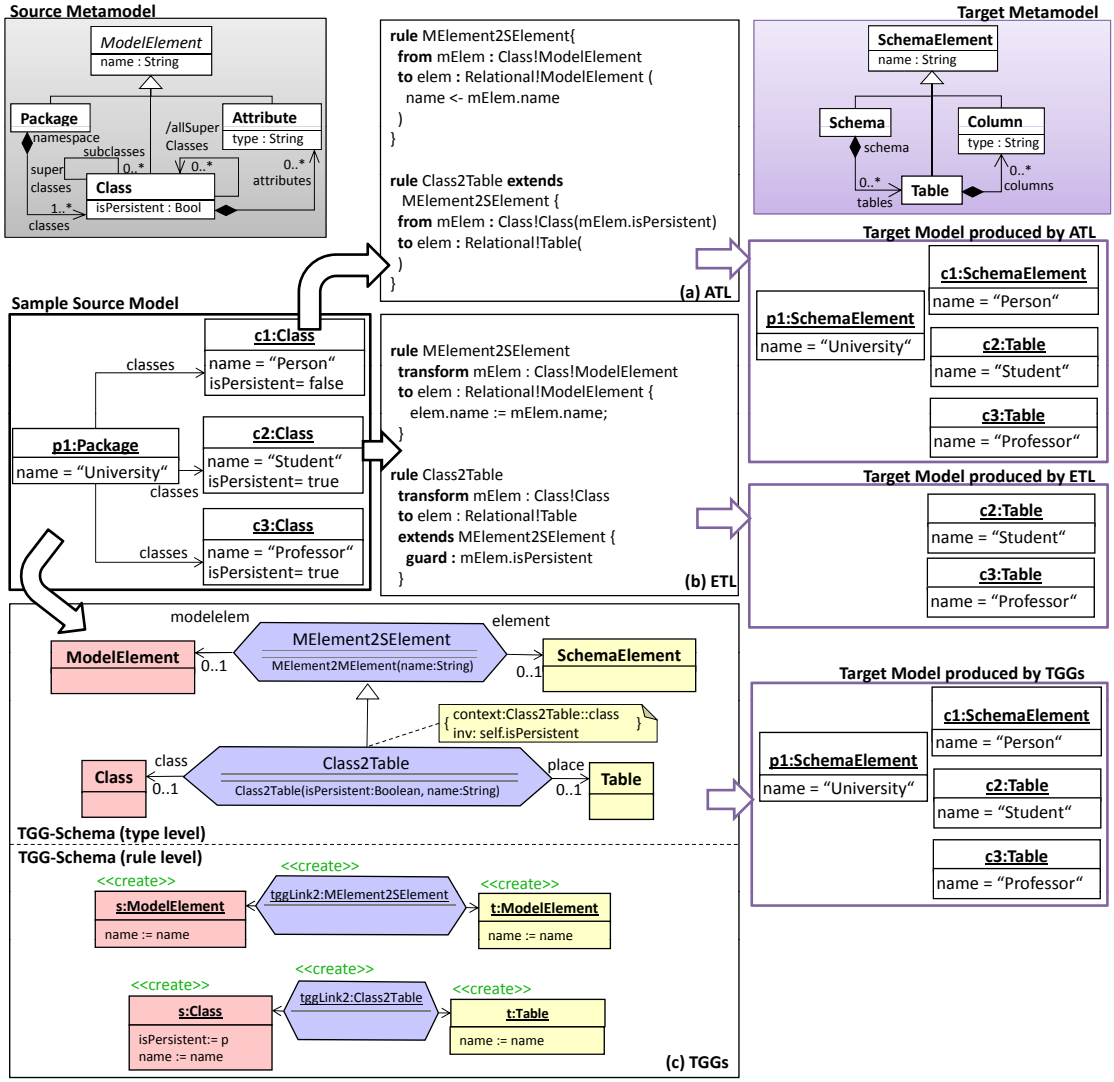


Figure 5.3: Transformation example in ATL, ETL and TGGs

5.2 Syntax

This subsection provides criteria for comparing transformation languages in terms of syntactic concepts that they support. For this, the metamodel presented in Fig. 4.2 is extended with inheritance related aspects as highlighted in Fig. 5.4. In the context of inheritance related aspects, three criteria are relevant. First, a `TransformationRule` may inherit from 1 or 1..* other transformation rules, depending on whether *single* or *multiple inheritance* is supported. Second, the concept of `abstract` rules may be supported in order to specify that a certain rule is not executable per se but provides core behavior that may be reused in subrules. One may distinguish between different *refinement modes* by which inherited parts are incorporated into

inheriting rules (modeled by the enumeration `RefinementMode` in Fig. 5.4). First, *override* implies that when a subrule refines an assignment of a superrule, the assignment of the subrule is executed together with those assignments in the superrule which are not overridden. In the refinement mode *inherit*, first, the overridden assignments are executed, and then the overriding assignment may alter the resulting intermediate result (such as by initializing some state by a supercall and then altering this intermediate result accordingly). Finally, the refinement mode *extension* induces that inherited assignments may not be changed at all. For consistency reasons, all assignments in a rule should follow the same refinement mode, therefore the refinement mode is specified on transformation rule level (cf. attribute `TransformationRule.mode`).

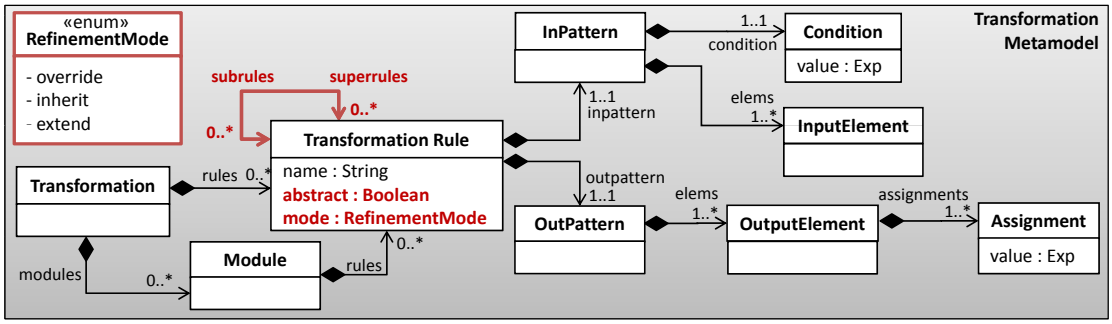


Figure 5.4: Inheritance-Related Concepts of Transformation Languages

5.2.1 Syntactical Comparison of Existing Languages

When comparing the considered languages (cf. Table 5.1), differences in the *number of allowed input elements* may be detected. Whereas ATL (multiple elements in `from` pattern) and TGGs (source object graph) allow several input elements to be bound to a rule, this is not possible in ETL (cf. single variable after `transform` keyword in Fig. 5.3). However, all of the languages evaluated support multiple *output elements* (multiple elements in `to` pattern in ATL and ETL, target object graph in TGGs). Although the *number of allowed input and output elements* is not directly related to inheritance, input and output elements play also a major role in inheritance, which is detailed in Section 5.3. Furthermore, all transformation languages allow for the specification of *conditions* (OCL expressions in ATL and TGGs, a `guard` in ETL, as can be seen in Fig. 5.3). ETL and TGGs support multiple inheritance, whereas ATL is restricted to single inheritance (keyword `extends` in ATL and ETL, inheritance arrow in type level of TGGs, cf. Fig. 5.3). All languages provide means to define abstract rules (keyword `abstract` in ATL, annotation `@abstract` in ETL, property `abstract` in TGGs). Finally, concerning potential *refinement modes of assignments*, none of the approaches evaluated provides specific keywords for explicitly choosing the semantics to be applied. Instead, ATL and ETL implicitly assume *override* semantics, and TGGs support the refinement mode *extension* since only new assignments may be added, but existing ones must not be modified.

In summary, all of the approaches evaluated support similar syntactical inheritance concepts. The main differences lie in the type of inheritance supported and the implicitly assumed refinement mode of assignments (cf. Table 5.1).

5.2.2 Inheritance Related Syntax in Transformation Nets

Based on the comparison above of today's transformation languages that support rule inheritance, in the following the according syntax in Transformation Nets is explained. The subpackage `DynamicElement` of the Transformation Net metamodel is accordingly extended as can be seen in Fig. 5.5. As already explained in Chapter 4, Transformation Nets allow for an *arbitrary* number of *input* and *output elements* which are represented by means of query or production patterns. This is specified in the metamodel by the two unbounded references `Transition.queryPatterns` and `Transition.productionPatterns`, as can be seen in Fig. 5.5. Furthermore, Transformation Nets allow for the specification of OCL conditions, as discussed in Subsection 4.4.2. Concerning inheritance specific aspects, Transformation Nets support *multiple inheritance*, which is specified in the metamodel in terms of the unbounded reference `Transition.superTransitions`. As can be seen in Fig. 5.6, the visualization of inheritance between transitions is equal to inheritance between classes, i.e., inheritance arrow between subtransition `Class2Table` and supertransition `MElement2SElement`. Finally, Transformation Nets allow to explicitly specify the *refinement mode of assignments* by means of the enumeration `RefinementMode` and the attribute `Transition.mode`, as can be seen in Fig. 5.5. As a default, override semantics is applied since it is prevalent in the evaluated transformation languages. The attributes `Transition.includeSubtypes`, which is used to influence the dynamic behavior (cf. Section 5.4) and `Transition.priority`, which may be used to solve ambiguous rule definitions (cf. Section 5.3) are specific to transformations, in order to overcome shortcomings of current mechanisms for rule inheritance in transformation languages, as detailed in the according sections.

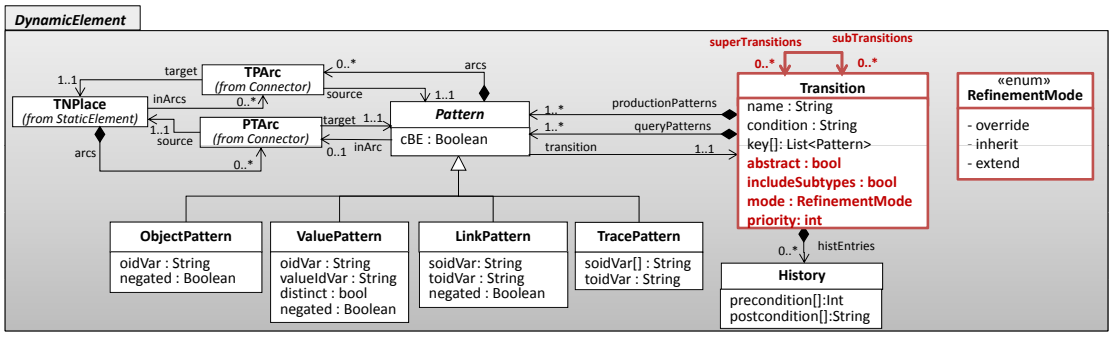


Figure 5.5: Extension of Transformation Net Metamodel to Represent Rule Inheritance

Fig. 5.6 shows the solution of the above example by means of Transformation Nets. The transition `MElement2SElements` translates objects that are instances of `ModelElement` to according `SchemaElement` instances and copies the value of `name` attribute. The subtransition `Class2Table` inherits from the supertransition `MElement2SElement`, whereby the object query pattern and the production query pattern `modelElement` are overridden. Thus, patterns of a supertransition may be overridden by equal variable identifiers. The remaining patterns of the supertransition are inherited, i.e., the subtransition `Class2Table` also queries for `name` attributes and produces according `name` attributes as may be seen by the tokens in the `name`

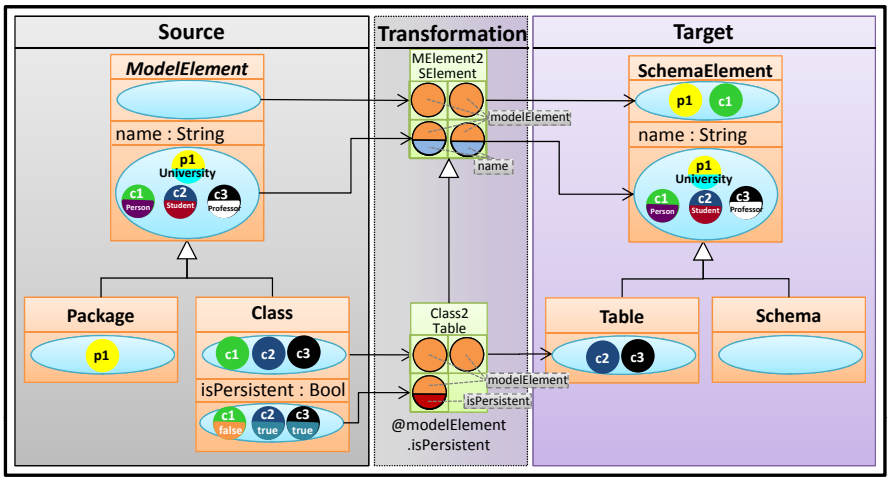


Figure 5.6: Example of Inheritance in Transformation Nets

place. Finally, the subtransition is extended by the additional query pattern `isPersistent` in order to query for persistent `Class` instances only. On inspecting the generated target model, one may see that only one `Table` instance is generated since only class `c1` is persistent. Nevertheless, since the token `c2` does not fulfill the condition of the subtransition, it may be matched by the supertransition and therefore the token `c2` is typed to `SchemaElement`, which is discussed in more detail in Section 5.4.

Table 5.1 summarizes the comparison of the syntactical comparison of inheritance related elements in current transformation languages as well as in Transformation Nets.

Table 5.1: Comparison of Inheritance Syntax

Rule Part	Values	ATL	ETL	TGGs	TN
Input Elements	1 | 1...*	1..*	1	1..*	1..*
Output Elements	1 | 1...*	1..*	1..*	1..*	1..*
Condition	Yes | No	Yes	Yes	Yes	Yes
Type of Rule Inheritance	Single | Multiple	Single	Multiple	Multiple	Multiple
Abstract Rules	Yes | No	Yes	Yes	Yes	Yes
Refinement Modes of Assignments	Override | Inherit | Extend	Override	Override	Extend	Override, Inherit, Extend

5.3 Static Semantics

In the previous subsection, criteria targeting the comparison of syntactic concepts have been identified. Now, criteria relevant for checking the static semantics of inheritance are elaborated. These criteria reflect the following semantic constraints: (i) incompatibility of input and

output elements of subrules and superrules in terms of type and number, (ii) non-instantiability of abstract classes, (iii) ambiguities in rule definitions, and (iv) conflicts in multiple inheritance.

Incompatibility of Input and Output Elements. In the context of transformation rules, both, feature assignments and conditions, should be inheritable to subrules. Thus, it has to be ensured that the *types* of the input and output elements of subrules have at least the features of the types of the elements of the superrule. Therefore, types of the input and output elements of a subrule might become more specific than those of the overridden rule. The inheritance hierarchy of the transformation rules usually follows the inheritance hierarchy of the MMs, i.e., a superrule always originates from a more general class the the subrule. Nevertheless, not for every class in the inheritance hierarchy of the metamodel a certain transformation rule needs to be provided. For example, if the source metamodel class `C` inherits from class `B`, and class `B` again inherits from class `A` it is allowed to specify only a rule `R1` that matches for instances of class `A` and a rule `R2` that matches for instances of class `C` and which inherits from rule `R1`. Nevertheless, here the question arises how to treat instances of class B, which is discussed in Section 5.4. For rule inheritance this means that co-variance for input and output elements is demanded, conforming to the principle of *specialization inheritance* in object-oriented programming [77]. This is in contrast to popular design rules for object-oriented programming languages, where a contra-variant refinement of input parameters and a co-variant refinement of output parameters of methods is required to yield type substitutability, also known as *specification inheritance* [99]. Additionally, the *number* of input and output elements should be extensible. In this respect, four cases of potential variations of input elements in type and number may be distinguished (cf. Fig. 5.7):

- **Same Number, Different Types (a).** As an example, Fig. 5.7(a) depicts the two rules, `RuleA2X` and `RuleB2Y`, that are bound to the source base classes `A` and `B` and to the target base classes `X` and `Y`, where both rules simply copy the contained features. Since source class `C` inherits from both classes `A` and `B` and the target class `Z` from the classes `X` and `Y`, the `RuleC2Z` may inherit from `RuleA2X` and `RuleB2Y`. Thus, the feature assignments of the superrules are reused (cf. grey assignments in Fig. 5.7(a)).

- **Same Number, Equal (Source or Target) Types (b).** This case (cf. Fig. 5.7(b)) may be counterintuitive, since inheritance is usually used to specialize some core behavior for subsets of instances, and subtypes are typically used to construct these subsets. In this case – at first sight – no subsets (according to *specialization inheritance*) are built, and it is unclear which rule should be executed for a combination of instances. Therefore, the required subsets must be built by applying corresponding disjoint conditions to the subrules in case of equal source types. In case the target type remains equal, feature assignments refer to target elements of the superrule. These scenarios occur if either the source or the target metamodel makes use of inheritance.

- **Different Number, Different Types (c).** Here, the subsets needed are built through the specialization of at least one input element (cf. Fig. 5.7(c)).

- **Different Number, Equal Types (d).** In this case, only the number of input or output elements is extended, but the types of elements bound in subrules remain the same. Thereby,

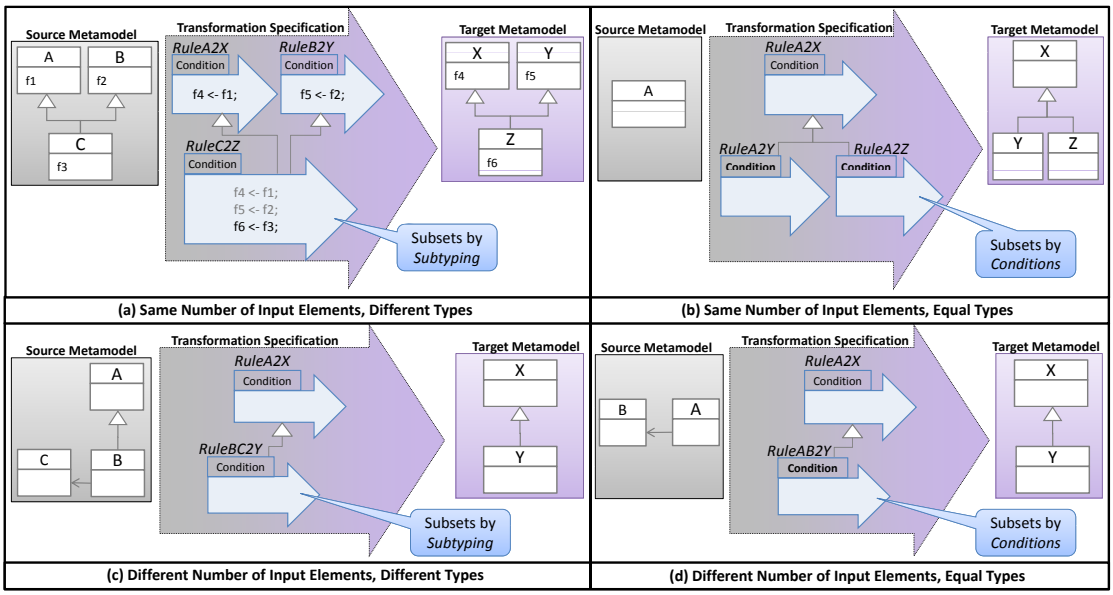


Figure 5.7: Rule Compatibility

the same problem as in case (b) arises, where the subsets must be realized by means of conditions which may require certain relationships between the matched input elements (cf. Fig. 5.7(d)).

One interesting question in the context of cases (b) and (d) is whether the instances that do not fulfill any of the conditions of the subrules are matched by the superrule (provided that the superrule is concrete). Since this question is closely related to dynamic semantics, this is further discussed in Section 5.4.

Non-Instantiability of Abstract Classes. Since abstract classes cannot be instantiated, it must be ensured statically that no concrete rule tries to create instances of an abstract target class as output. Only abstract rules are allowed in this case, since they are not executed themselves but must be refined by a subrule. The situation is different for abstract source classes: although an abstract source class may not have any direct instances, indirect instances may be affected by the transformation rule.

Ambiguities in Rule Definitions. An ambiguity between inheriting transformation rules may arise if a rule requires multiple input elements, and if there is no single rule for which the match in runtime types is closer than all the other rules. This is analogous to the problem that arises in multiple dispatching as needed for multi-methods (cf. [1, 31]), since choosing a method requires the run-time type not of a *single* input element, but of a *set* of input elements. Thus, the method whose run-time types most closely match the statically specified types should be dispatched at run-time. A simple example of such a problem is depicted in Fig. 5.8(a). Three transformation rules are specified taking two input elements of different metamodel types, respectively. Now, suppose that a pair of instances `(b,y)` of type `B` and `Y` is transformed, and assume that the rules might also match indirect instances. The transformation engine should

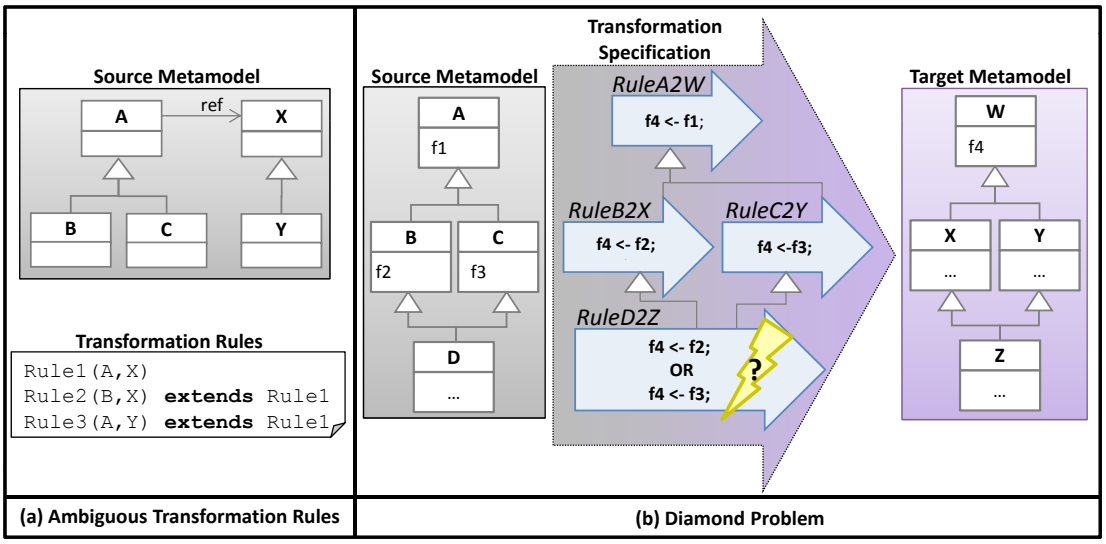


Figure 5.8: Examples of Static Constraints: (a) Rule Ambiguity and (b) Diamond Problem

now look for a rule whose arguments *most closely match* the pair `(b,y)`. In this case, no single rule may be determined, since `Rule2` and `Rule3` are equally good matches. Thus, the set of defined transformation rules is ambiguous.

Conflicts in Multiple Inheritance. The *diamond problem* [148], also referred to as *fork-join* inheritance [132], arises, when contradicting assignments are inherited via different inheritance paths. Consider, for instance, the common superrule `A2W` in Fig. 5.8(b), which contains an assignment for copying a feature value. This assignment is overridden within `RuleB2X` and `RuleC2Y`. Thus, it may not be decided in `RuleD2Z` which assignment should be applied, unless assistance is given by the transformation designer.

5.3.1 Comparison of Static Semantics of Existing Languages

This part of the comparison evaluates in how far the static semantics of inheritance is checked in each transformation language (cf. Table 5.2). Concerning *input and output elements*, in ATL a violation of co-variance is detected at run-time, since missing features result in a "feature not found" exception. In ETL no error is reported, which leaves the detection of the resulting erroneous instances to the transformation designer or another model management operation executed after the transformation. In TGGs this results in a compile-time error in the upcoming implementation, since the main principle is that applying the subrule should guarantee the existence of the subgraph created by the superrule. Concerning the number of input elements, in ATL a run-time error occurs, if the number is changed in any way (including name changes). Thus, ATL requires that the number of input elements remains the same. In contrast, ATL does not raise any exception if the number of output elements is restricted, since they are produced even if they are not re-specified. In ETL, this criterion is not applicable, since ETL restricts the number of input elements to exactly one anyway. In ETL a run-time error ("index out of bound"

exception) is raised if the number of output elements is restricted. In TGGs – to conform to the main principle that applying the subrule should guarantee the existence of the subgraph created by the superrule – only an extension of the number of input and output elements is allowed, which is again going to be ensured statically in the upcoming implementation.

None of the languages evaluated detect *concrete rules referencing abstract classes* at compile-time, throwing run-time errors instead. Concerning *ambiguous rule definitions* ATL does not throw any exceptions – neither at compile-time nor at run-time. Instead, the first matching rule defined in the transformation specification is executed. In ETL, the problem of ambiguous rule definitions may not arise, since multiple input elements are not supported. In TGGs, a run-time error is thrown. The *diamond problem in multiple inheritance* does not apply to ATL, since multiple inheritance is not supported. Although the diamond problem is detected in ETL and TGG at compile-time, it is checked on a coarse-grained level only, i.e., diamonds that do not include ambiguous assignments also cause errors.

In summary, static inheritance checks are poorly supported by ATL and ETL. In ATL, none of the static semantics are checked statically. The same is true for ETL with the exception of the diamond problem. In contrast, TGGs at least conceptually propose quite a number of static checks that will be considered in the upcoming implementation of rule inheritance.

5.3.2 Static Semantics in Transformation Nets

As discussed before, support for checking the static semantics is limited. This gives rise to run-time errors or – even worse – to erroneous target instances with no error message. Thus, the tedious task of checking the static semantics is left to the transformation designer. Since Transformation Nets are specified on basis of a metamodel, OCL invariants may be employed in order to ensure the required static semantics and to detect invalid configurations concerning rule inheritance, helping the transformation designer in detecting defects. Consequently, the above presented requirements are specified as invariants over the metamodel depicted in Fig. 4.3 on page 73, which are explained in detail in the following and summarized in Table 5.2.

Incompatibility of Input and Output Elements. Transformation Nets allow the transformation designer to change the input and output elements in both, number and type. It is allowed that a subtransition either extends the number of input and output elements. Since the behavior specified in supertransitions may only be extended but not restricted (following the common principle in object oriented programming), those input or output elements that are not re-specified in the subtransition are considered nevertheless, i.e., they are inherited. Furthermore, the types might be overridden in a co-variant manner only, which is ensured by the OCL invariant depicted in Listing 5.1 (shown for query patterns only, since it is analogous for production patterns). For this, first all `ObjectPatterns` are selected from a transition's query patterns (cf. line 6 in Listing 5.1). For this, a derived attribute `queryObjectPatterns` is used (cf. line 1 - 3 in in Listing 5.1). For every pattern it is checked if a pattern overrides a pattern of any supertransitions. In order to get all super transitions a derived property `allSuperTransitions` is specified for transition instances (cf. Listing 5.2 and line 8 in Listing 5.1). If the pattern variable of the subtransition is contained in the set of pattern variables from the supertransitions (cf. line 9 in Listing 5.1) the pattern overrides the basic behavior in the supertransition. In order to ensure a covariant relationship in this case, the according

source `Class` places have to be in an inheritance hierarchy, i.e., the source class of the pattern of the subtransition is a subclass of the source class of the pattern in the supertransition. If the class referred by the pattern of the supertransition is contained in the set of all super classes of the class referred by the pattern in the subtransition, the invariant is fulfilled. (cf. line 10 - 13 in Listing 5.1).

Listing 5.1: Invariant to Check Covariant Overrides

```
1  context Transition: def queryObjectPatterns: Set(ObjectPattern)=
2  self.queryTokens->select(x | x.oclIsTypeOf(QueryObjectPattern)) --select object patterns
3    ->collect(x | x.oclAsType(QueryObjectPattern)) --cast them
4  ------------------------------------------------------------------------------
5  context Transition inv Covariance:
6  self.queryObjectPatterns->forAll(qop : QueryObjectPattern |
7        -- select query tokens of all super transitions
8       self.allSuperTransitions->collect(t : Transition | t.queryObjectPatterns)
9         ->collect(oidVar)->includes(qop.oidVar)) implies -- if pattern is overridden
10          qop.inArc.source.allSuperClasses->includes( -- check covariance condition
11          self.allSuperTransitions->collect(t:Transition | t.queryObjectPatterns)
12          ->flatten()->any(sqop : QueryObjectPattern |
13                sqop.oidVar = qop.oidVar).inArc.source))
```

Listing 5.2 shows the OCL expression for calculating the set of all super transitions. Thereby the supertransitions of a certain transition are unified with their supertransitions (by means of a recursive call).

Listing 5.2: Derived Attribute to Calculate Transitive Closure of Transitions

```
1  context Transition: def allSuperTransitions: Set(Transition)=
2    self.superTransitions->asSet()->union(self.superTransitions->
3      collect(s| s.allSuperTransitions)->asSet())
```

Non-Instantiability of Abstract Classes. Following the rules presented above it should be forbidden that a concrete transition targets an abstract class. In this respect, an OCL invariant first checks if the transition is abstract or not as can be seen in line 2 in Listing 5.3. If the transition is concrete first all production `ObjectPatterns` are collected. For this, a derived attribute `getEffectiveProductionObjectPatterns` (cf. line 1 in Listing 5.3) is provided which collects the set of production patterns, i.e., the patterns of the actual transitions and all patterns of according super transitions which are not overridden. Please note that for this task again a derived attribute getVars is defined which returns the set of the actuals variables according to the type of the production patterns, e.g., `oidVar` in case of an `ObjectPattern` or `oidVar` and `valueIdVar` in case of a `ValuePattern`. Then the according outgoing arcs are selected and it is checked if all arcs target abstract classes (cf line 14 in Listing 5.3).

Listing 5.3: Invariant to Check Non-Instantiability of Abstract Classes

```
1  context Transition inv getEffectiveProductionObjectPatterns : Set(Pattern) =
2  self.productionPatterns->union(
3        -- select query tokens of all super transitions
4       self.allSuperTransitions->collect(t : Transition | t.productionObjectPatterns)
5       --select non overridden patterns
6       ->collect(getVars)->excludes(self.productionPatterns.getVars))
7  ------------------------------------------------------------------------------
8  context Transition inv NoAbstractTargetClassForConcreteRule:
```

```
9  (not self.abstract) implies -- if transition is concrete
10   -- select and cast object patterns
11   self.getEffectiveProductionPatterns() -- get all patterns
12     ->collect(outArcs)-> flatten()
13       --check if the target class is concrete
14        -> forAll(x:TPArc | not x.target.abstract)
```

Ambiguities in Rule Definitions. In order to check for ambiguous rule definitions, an OCL invariant is provided that makes use of the derived operation `isAmbiguous()`, which is called if a certain transitions exhibits more than one query object patterns and if more than one subtransition exists, since only in this case ambiguities may arise (cf. Listing 5.4).

Listing 5.4: Invariant to Check Ambiguous Rule Definitions

```
1 context Transition inv RuleAmbiguity:
2 self.allSuperTransitions->collect(queryObjectPatterns)->size() > 1 and
3 self.superTransition->collect(subTransition)->size() > 1 implies
4     self.isAmbiguous()
```

The derived operation `isAmbiguous` is implemented in Java since for this check a complex data structure is needed. In the following, the basic idea of the derived property is described on basis of the example presented in Fig. 5.8(a). In order to check the invariant, first the root of the inheritance hierarchy in the transitions is searched, i.e., `Rule1` in the exemplary realization in Transformation Nets shown in Fig. 5.9(a). Next, the according source classes of the root transition's object patterns are collected, i.e., `A` and `X` in our example. Since type substitutability should be considered, all subclasses are collected additionally, i.e., `B` and `C` for `A` and `Y` for `X`, which are stored in an according matrix (cf. Fig. 5.9(b)). By building all potential combinations of the input parameter, the most specific rule that is applicable has to be found, following the *argument subtype precedence* principle presented in [1]. For example, `Rule1` would be chosen for instances of classes `A` and `X`, since objects required by the transition and the matched objects are equally typed. If, for example, instance of classes `C` and `Y` are considered, `Rule1` and

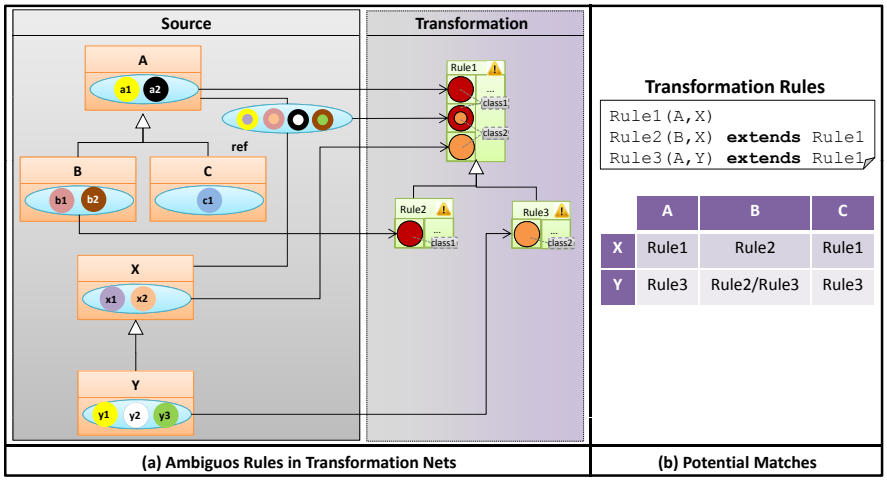


Figure 5.9: Transformation Example in Transformation Nets

`Rule3` would theoretically be applicable. Nevertheless, if no exact match is found (i.e., not all parameters demanded are equally typed as the matched objects), the distance in the inheritance hierarchy is calculated, i.e., since *B* is a direct subclass of `A`, the distance of `B` to `A` would be 1, whereas a direct subclass of `B` would have a distance of 2 to `A` and so on. When trying to match `Rule1`, the distance of instances of classes `C` and `Y` is 2, whereas the distance of `Rule3` is only 1. Therefore, in this case `Rule3` is preferred. However, if instances of classes `B` and instance of classes `Y` should be matched, *Rule1* and *Rule2* exhibit both a distance of 1. Thus, it is undecidable which transition to choose. In this case, the derived property returns false which leads to a warning in the Transformation Net. In order to resolve this problem, the transformation designer may either specify an according transition or he may make use of the priority flag of the transitions (cf. attribute `Transition.priority` in Fig. 5.5, specifying which transitions should be preferred, whereby a lower value indicates precedence.

It must be noted that considering the research field of multi-methods in object oriented programming, there are approaches for explicit disambiguation (e.g., [2] proposes a minimal set of method redefinitions necessary for disambiguation) which could be reused in transformation languages. However, this is not the focus of this thesis.

Conflicts in Multiple Inheritance. Although both languages that support multiple inheritance (ETL and TGG) check the diamond problem statically, the only check is – as already mentioned – if there is a fork-join path in the inheritance hierarchy but not if contradicting assignments are inherited via these different inheritance paths, i.e., no fine grained checks are provided. In contrast to that, the OCL invariant specified for Transformation Nets checks the diamond problem on this fine grained level. For this, it has nevertheless to be checked if a diamond exists in the inheritance hierarchy in a first step. As shown in Listing 5.5 first the supertransitions are collected, once allowing duplicates (cf. derived attribute `allSuperTransitionsWithDuplicates` – lines 1 to 3 in Listing 5.5), and once prohibiting duplicates. If the size of these collections differs, this means that there exists a diamond, i.e., at least one superclass may be reached by different paths in the inheritance hierarchy. A diamond does not necessarily lead to a conflict but if and only if transitions are found on the same level in the inheritance hierarchy which target the same feature as depicted in Fig. 5.8(b). To check this, all supertransitions which exhibit multiple inheritance (cf. line 9 in Listing 5.5) are selected. From these transitions the according targets of the `AttributePatterns`, i.e., the according attribute places, are selected twice, whereby once duplicate attributes are removed. For this again a derived attribute `productionAttributePatterns` is specified, which collects the according production attribute patterns. If the size of the collections differs, at least two `AttributePatterns` target the same attribute which leads to an undecidable situation, i.e., it is undetermined which transition may set the according attribute value. Therefore, the invariant fails in this situation.

Analogously to the explicit disambiguation of ambiguous rule definitions, the transformation designer could be supported by proposals which assignments must be overridden in rules in order to achieve unambiguous assignment definitions.

Listing 5.5: Invariant to Check Diamond Problem

```
1 context Transition: def allSuperTransitionsWithDuplicates: Set(Transition)=
2   self.superTransitions->asBag()->union(self.superTransitions->
```

```
3       collect(s| s.allSuperTransitionsWitDuplicates)->asBag())
4   ------------------------------------------------------------------------
5   context Transition inv NoDiamond:
6   self.allSuperTransitionsWithDuplicates->size() >
7       self.allSuperTransitions->size() implies
8       --select transitions with multiple inheritance
9       not (self.allSuperTransitions->select(t:Transition | t.subTransitions->size() > 1)
10      --collect all target attributes of production attribute patterns
11        ->forAll(y : Transition | y.subTransition->collect(productionAttributePatterns)
12        ->collect(outArcs)->collect(target)->size() >
13        --again collect the target attributes of the patterns but remove duplicates
14        y.subTransition->collect(productionAttributePatterns)
15        ->collect(outArcs)->collect(target)->asSet()->size())
```

Table 5.2 summarizes the comparison of static semantics in current transformation languages as well as in Transformation Nets.

Table 5.2: Comparison of Static Semantics with respect to Inheritance

Verification Target	Fault	Values	ATL	ETL	TGGs	TN
Input Elements	Non-co-variant Type Change	[Compile-Time|Run-Time|No] Error	Run-Time Error	No Error (invalid target model)	Compile-Time Error	Compile-Time Error
	Restriction in Number	[Compile-Time|Run-Time|No] Error	Run-Time Error (also with extension)	n.a. (cf. syntax)	Compile-Time Error	n.a. (base types need not be respecified)
Output Elements	Non-co-variant Type Change	[Compile-Time|Run-Time|No] Error	Run-Time Error	No Error (invalid target model)	Compile-Time Error	Compile-Time Error
	Restriction in Number	[Compile-Time|Run-Time|No] Error	n.a. (output elements are still produced even if not specified again)	Run-Time Error	Compile-Time Error (except of output to input modification)	n.a. (output elements are still produced even if not specified again)
Abstract Target Classes	Concrete Rules for Abstract Target Classes	[Compile-Time|Run-Time|No] Error	Run-Time Error	Run-Time Error	Run-Time Error (application fails)	Compile-Time Error
Rule Ambiguity		[Compile-Time|Run-Time|No] Error	No Error (first matching rule in file wins)	n.a. (cf. syntax)	Run-Time Error	Warning at Compile-Time (Transition with lowest priority wins)
Diamond Problem		[Compile-Time|Run-Time|No] Error	n.a. (cf. syntax)	Compile-Time Error	Compile-Time Error	Warning at Compile-Time

5.4 Dynamic Semantics

After discussing the static semantics the focus is shifted to the dynamic semantics, i.e., how transformation specifications may be interpreted at run-time. In this context, two main aspects are investigated: (i) which rules apply to which instances, i.e., *dispatch semantics* and (ii) how a set of inheriting rules gets executed, i.e., *execution semantics*.

Dispatch Semantics. In order to execute transformation specifications, it must be determined which rules apply to which instances, i.e., transformation rules must be dispatched for source model instances. In [38], potential strategies and scheduling variations of rules were discussed, but without any focus on inheritance. Thus, literature on dispatching rule in model transformation does not indicate whether *type substitutability* should be considered. This principle is well-known in object-oriented programming and states that, if S is a subtype of T, objects of type T may be safely replaced by objects of type S [99]. Type substitutability for transformation rules would mean that if a rule may be applied to all instances of class T, then this rule may also be applied to all instances of all subclasses of T. Consequently, if no specific subrule

is defined for instances of a subclass, then these instances of the subclass may be transformed by the rule defined for the superclass.

Concerning the evaluation of the condition, two main strategies may be followed during dispatching. First, the condition may be considered as part of the matching process, i.e., if the condition fails, the rule is not applicable, but a superrule might be applied (*rule applicability* semantics). Second, the condition is not considered in the matching process, i.e., matching takes place on the specified types of the input elements only and thus, those elements, which do not fulfill the condition, are filtered, but never matched by a superrule anymore (*filter* semantics).

Execution Semantics. After having determined which rules are applicable to which source model instances, the question arises how a set of inheriting rules is executed. A first distinguishing criterion is whether the concept of inheritance is directly supported by the execution engine or whether the inheritance hierarchy is first flattened to ordinary transformation code in a pre-processing step. Independent of whether the inheritance hierarchy is flattened or not, various strategies may be applied to evaluate conditions and to execute assignments. This raises questions such as "Are conditions of a superrule also evaluated?" and "Are the assignments of a superrule executed before the assignments of a subrule". Hence, the main characteristics of executing methods in an inheritance hierarchy in object-oriented programming [148] are investigated: (i) the *completion of the message lookup*, i.e., whether only the first matching method is executed (*asymmetric*) or all matching methods along the inheritance hierarchy are executed (*composing*), and (ii) the *direction of the message lookup*, i.e., whether a method lookup starts in the subclass (*descendant-driven*) or in the superclass (*parent-driven*).

5.4.1 Comparison of Dynamic Semantics of Existing Languages

In order to compare the dynamic semantics, *dispatch* and *execution semantics* are investigated (cf. Table 5.3). Considering *dispatch semantics*, one may see that the output models produced by ATL and TGGs (Fig. 5.3(a) and (c)) include only two `Table` instances, since only `Class c2` and `c3` fulfill the specified condition in the subrule. As ATL and TGGs support *type substitutability* and *rule applicability* semantics for conditions, instance `c1` is matched by the more general superrule `MElement2MElement`, and therefore creates the target `ModelElement c1`. Due to *type substitutability*, the indirect instance `p1` is matched by the superrule, and therefore the target `ModelElement p1` is created. In contrast, ETL does not support *type substitutability* by default. Thus, although the specifications in ETL and ATL are syntactically very similar, the produced target models differ. ETL's target model contains only the two `Table` instances `c2` and `c3`, produced by the rule `Class2Table`. The dispatch semantics may be modified by annotating rules with `@greedy` in ETL. This means that such rules also match indirect instances, but the interpretation is different than in ATL and TGGs, since the superrule still regards all instances irrespective of whether the instances have already been matched by subrules or not. Adding the `@greedy` annotation to the rule `MElement2MElement` in our example would therefore create six instances in total: four `SchemaElement c1, c2, c3, p1` instances produced by the superrule `MElem2MElement`, and two `Table` instances `c2` and `c3` produced by the subrule `Class2Table`. Even if type substitutability is enabled in ETL, the result of the condition evaluation does not influence the dispatch semantics because the super-

rule always matches all direct and indirect instances, disregarding specialized subrules. Thus, the condition semantics is evaluated as not applicable in ETL.

Regarding inheritance support within the execution engine, in ATL inherited rules are flattened during compilation and may thus use optimization strategies, i.e., the ATL compiler inlines the assignments of a superrule. In contrast, ETL supports inheritance within the execution engine, which reduces the amount of code generated. In TGGs, this criterion is not applicable, since an inheriting TGG rule contains a copy of the superrules, which causes code duplication. Concerning the evaluation of conditions, all compared transformation languages exhibit a *composing* completion of the lookup, i.e., an instance processed by a subrule must fulfill all the specified conditions up the inheritance hierarchy (i.e., and conjunction). The actual evaluation is parent-driven in ATL and descendent-driven in ETL but non-applicable in TGGs, since a subrule lists all its inherited conditions. All approaches execute *all* assignments along the inheritance hierarchy (i.e., *composing* completion of the lookup). Finally, the direction of lookup in assignments occurs descendent-driven in ATL and parent-driven in ETL. Thus, in ATL (i) the assignments of the superrule, which are not overridden, (ii) the overridden assignments, and (iii) new assignments specified in the subrule are executed realizing the optimization strategy. In contrast, in ETL, (i) the assignments of the superrule and (ii) the assignments of the subrule are executed. In TGGs this is again not applicable. More specifically, TGGs enforce composition already in the syntax, which causes code duplication.

In summary, the main difference in terms of dynamic semantics lies in the application of type substitutability, which is user-defineable in ETL, but interpreted in a different way than in ATL and TGGs. ETL has the disadvantage that several target instances for a single source instance are created when a superrule is annotated with `@greedy`. Moreover, all of the transformation languages implement a *composing* behavior for conditions and assignments. Thus, the lookup direction does not influence the result of the transformation.

5.4.2 Dynamic Semantics in Transformation Nets

As the example in Fig. 5.3 reveals, similar syntax (cf. ATL and ETL) does not necessarily lead to the same results, which implies different dynamic semantics. This is undesirable, since the dynamic semantics is not made explicit by any syntactical elements to the transformation designer. Thus, the transformation designer must know the design decisions taken in each transformation language in order to obtain the desired result. The current situation concerning rule inheritance is comparable to the situation in the early stages of object-oriented programming, where no common agreements on the dynamic semantics of inheritance had been reached. This emphasizes the need to consider rule inheritance in the runtime model in order to make the taken design decisions explicit. Furthermore, since different dynamic semantics are supported by the compared languages, Transformation Nets should allow to alter the dynamic semantics.

Considering *dispatch semantics*, Transformation Nets employ per default type substitutability, similar to ATL and TGGs. Nevertheless, in order to also support transformation languages that do not make use of type substitutability, the transformation designer might change this behavior by setting the boolean flag `Transition.includeSubtypes` (cf. Fig. 5.5) to false. If the flag is set to false, then a behavior equal to ETL (without the `@greedy` annotation) results,

i.e., only direct instances are transformed but not indirect instances[2]. Since those approaches that support type substitutability all exhibit a rule application semantics of conditions, Transformation Nets do so as well. Therefore, the example depicted in Fig. 5.6 exhibits the same target model as ATL and TGG did (cf. Fig. 5.3), i.e., the persistent classes `c2` and `c3` may be transformed by the transition `Class2Table` and thus, only two target elements are typed to `Table` whereas the remaining elements are transformed by the base rule `MElement2SElement` and are therefore typed to `SchemaElement` only (cf. Fig. 5.6).

Regarding *execution semantics*, the inheritance support is flattened during compilation since the underlying CPNs do not provide any means to deal with inheritance. Concerning the evaluation of conditions, Transformation Nets exhibit a composing behavior. In this respect, all conditions along the inheritance hierarchy must be fulfilled since they are concatenated by a logical *and*. The actual evaluation thereof is done in descendent-driven manner since first the condition of the subrule is evaluated before the conditions of superrules are evaluated. Concerning assignments, Transformation Nets follow a composing strategy as all other approaches do. Depending on the actual refinement mode, different strategies are used. If the refinement mode `override` is selected, the assignments of superrules which are not overridden in a subrule are considered together with the overridden and newly added assignments of a subrule, equal to ATL (cf. Table 5.3). If the mode `extend` is selected, the strategy is the other way round, i.e., overridden assignments are not considered, but only the newly added assignments together with those of the superrules are considered. Furthermore, this leads to a further static constraint which raises an error in case a subtransition overrides a pattern of the supertransition. Finally, if the refinement mode `inherit` is selected, the assignments of the overridden pattern are copied to the overriding pattern of the subtransition and are executed together with the assignments of overriding pattern. When having a look at the direction of lookup for assignments this criteria is not applicable since all the assignments are aggregated during compilation and thus, it is undecidable which assignments are executed first.

Table 5.3: Comparison of Dynamic Semantics of Inheritance

Criterion		Subcriterion	Values	ATL	ETL	TGGs	TN
Dispatch semantics		Type Substitutability	Yes | No	Yes	User-Definable (default no)	Yes	User-Definable (default yes)
		Condition Semantics	Filter | Rule Applicability	Rule Applicability	n.a.	Rule Applicability	Rule Applicability
Execution Semantics	Inheritance Support	-	Flattened | Direct engine support	Flattened	Direct engine support	n.a. (since flattened in patterns already)	Flattened
	Condition	Completion of lookup	Asymmetric | Composing	Composing	Composing	Composing (by copy)	Composing
		Direction of lookup	Parent-driven | Descendent-driven	Parent-driven	Descendent-driven	n.a.	Descendent-driven
	Assignments	Completion of lookup	Asymmetric | Composing	Optimized Composing	Composing	Composing (by copy)	Optimized Composing
		Direction of lookup	Parent-driven | Descendent-driven	Descendent-driven	Parent-driven	n.a.	n.a. (aggregated during compilation)

[2]The dynamic semantics of ETL's `@greedy` is not explicitly considered, but could be simulated by independent transitions which do not inherit from each other. Thereby, the transition that represents the transition for the according ETL subrule has to incorporate the assignments of the superrule.

5.5 Summary

In summary, this chapter has presented different inheritance concepts between transformation rules. Thereby, three different dimensions have been considered, being (i) *syntactic aspects*, i.e., which language constructs are needed to express inheritance between transformation rules, (ii) *static semantics*, i.e., whether a set of inheriting transformation rules is well-formed at compile-time and (iii) *dynamic semantics*, i.e., how inheriting rules interact at run-time. To identify the concepts that should be represented by Transformation Nets, current transformation languages supporting rule inheritance were analyzed. Based on these findings, the realization of rule inheritance in Transformation Nets was discussed. Although rule inheritance is only considered by a few declarative model-to-model transformation languages (ATL, ETL, and TGGs) the inclusion of inheritance concepts into the runtime model (i) broadens the scope of applicability of the runtime model and (ii) reveals differences in the semantics of current languages, which are not obvious at first sight.

The previous two chapters introduced the concepts considered by Transformation Nets. Furthermore, it was stated that Transformation Nets represent a DSL on top of CPNs, hiding the actual details thereof from the transformation designer. Nevertheless, in order to make use of efficient execution engines available for CPNs and their formal execution semantics as well as their properties, Transformation Nets may fully be compiled into standard CPNs, which is presented the following chapter.

Chapter 6

Colored Petri Nets as Semantic Domain for Transformation Nets

Mathematics is the language in which
God has written the universe.

— Galileo Galilei

Contents

The previous chapter introduced the concepts of Transformation Nets. The focus was rather on the syntactic concepts provided, but their actual semantics was only informally presented. To define the meaning of the syntactical concepts, a mapping to a semantic domain is required following the authors of [63] who state that "any language definition must consist of the syntax, the semantic domain and a semantic mapping from the syntactic concepts to the semantic domain". Consequently, a semantic domain must provide means to make the semantics of the syntactic constructs explicit. Since Transformation Nets represent a DSL on top of CPNs, this chapter presents how CPNs may be used as a semantic domain for Transformation Nets. Thereby, (i) the basics of Petri Nets are introduced by explaining the core concepts, which form the basis for higher-level Petri Nets such as CPNs. The concepts are first introduced informally and second, the formal definition is provided. After that, (ii) the concepts of Transformation

Nets are formalized as well, in order to show that these concepts may be expressed by means of CPN concepts. This formalization builds the basis for (iii) the compilation of Transformation Nets into CPNs, whereby first the compilation of places and tokens, i.e., the static parts of Transformation Nets, and second, the compilation of transitions, patterns and conditions is shown. Finally, (iv) the compilation of rule inheritance and modules is discussed.

6.1 Introduction to Petri Nets and Colored Petri Nets

In general, Petri Nets and CPNs provide formal means to model, execute and analyze systems [72,118]. The concepts of CPNs provide the semantic domain for the specified Transformation Net DSL, i.e., the concepts of Transformation Nets may be compiled into concepts of CPNs. Basically, Petri Nets and CPNs exhibit the following characteristics [124], which make them favorable as a semantic domain:

- **Concise Set of Language Elements.** Petri Nets in its simplest form consist of places, transitions and tokens, only. By means of simple rules, transitions may fire and thus stream tokens from one place to another. In this respect, since it is possible to represent metamodels as places, the transformation logic as transitions and the models as tokens. The actual transformation logic may be followed in a process-oriented manner. In general the formalism is easy to understand, thus fostering understandability of model transformations.

- **Formal Basis.** The formal mathematical basis of Petri Nets allows to calculate properties which may be used to verify Petri Nets. Since Transformation Nets base on CPNs these properties may be applied in the domain of model transformations as well in order to verify a transformation specification, i.e., it might be checked if the specified transformation terminates or if it is confluent (cf. Section 7.4).

- **Generality.** Petri Nets may be used to model a broad variety of systems, especially, to model parallel systems [124]. Since many transformation languages do not specify a certain order concerning transformation rules, i.e., it may be undefined which rule might fire first, Petri Nets could make these parallelism explicit. Furthermore, the concepts build the basis for several different modeling languages such as UML Activity Diagrams. Therefore, Petri Nets are well-known in the domain of software engineering.

- **Graphical Syntax.** The graphical representation allows a comprehensible visualization of static and dynamic aspects of the modeled systems. Nevertheless, the specification of complex inscription expressions in high-level Petri Nets, typically in a language a transformation designer is not familiar with, makes the syntax unsuitable for direct use to specify model transformations. Therefore, Transformation Nets introduced a DSL on top of Petri Nets, aligning the Petri Net syntax to the domain of model transformations.

- **Tool Support.** There are sound tools available to specify, execute and verify Petri Nets, building the basis for the prototype presented in Chapter 8.

In order to introduce the concepts of Petri Nets, first the basics of Petri Nets are discussed. Afterwards Colored Petri Nets are presented, which extend the basic concepts of Petri Nets, in order to establish a common understanding of the underlying principles.

6.1.1 Petri Nets in a Nutshell

Petri Nets describe a bipartite, directed graph. *Places*, represented as ovals, and *Transitions*, represented as rectangles, form the nodes and *Arcs* form the edges (cf. Fig. 6.1(a)) of the graph. Arcs always connect a place and a transition, but never two places or two transitions. The places from which arcs originate are called *input places* of a transition whereas places in which arcs from a transition end are called *output places* of a transition. Places contain *tokens* whereby the accumulation of all tokens in all the places is called *marking*. The initial marking thereby denotes the initial token arrangement. Tokens may enable transitions if all input places of a transition contain at least the number of tokens as required by the *weights* on the according arcs and if the *capacity* of the target places are not exceeded after firing. If a transition is enabled, it is allowed to fire. If a transition fires, it consumes the tokens of the input places and produces tokens in the output places (cf. Fig. 6.1(a)).

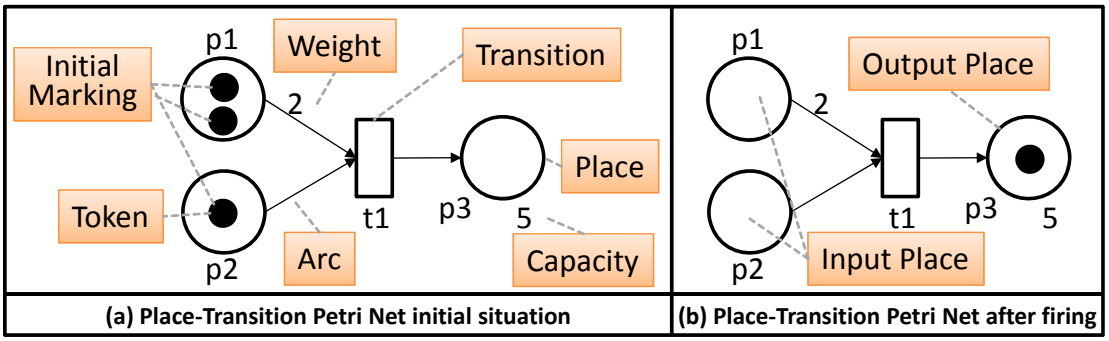


Figure 6.1: Simple Place-Transition Petri Net

Formal Definition of Static Parts. To be more formal the definitions will be precised, thereby resembling the definitions in [124]. A Petri Net Graph may be described as a 3-tuple (P, T, A), where P is a finite set of places and T is a finite set of transitions. P and T are disjoint subsets ($P \cap T = \emptyset$) meaning that no element may be both, a place *and* a transition. The set of arcs A is the cross product of places P and transitions T and vice versa, i.e., $A \subseteq P \times T \cup T \times P$. Furthermore, places may contain so-called *tokens*. Thereby, a marking is a function M that assigns a number of tokens to each place, i.e., $M\colon P \to \mathbb{N}$. Please note that in the simplest form of Petri Nets, so-called *Condition-Event-Nets (CEN)*, places are only allowed to contain either exactly one or zero tokens. In addition, *Place-Transition-Nets (PTN)*, which are seen as a synonym to the general term Petri-Net, allow to assign a capacity C to places, i.e., $C\colon P \to \mathbb{N}_\infty$ and a weight W to arcs, i.e., $W\colon A \to \mathbb{N}$. An unbounded capacity ∞ is the default, in case that no capacity is given for places and a weight of one is the default for arcs. Thus, a CEN may be seen as a PTN with a constant capacity and weight of one, i.e., $C = 1, W = 1$. Consequently,

a Petri Net Graph (P, T, A) is extended to a 6-tuple (P, T, A, C, W, M) whereby the initial number of tokens has to be less or equal to the allowed capacity, i.e., $\forall p \in P\colon M(p) \leq C(p)$.

Formal Definition of Dynamic Parts. The task of a transition is then to consume tokens from the input places and produce tokens in the output places. Thus, firing a transition t in a marking M consumes $W(A(p,t))$ tokens from each of its input places p, and produces $W(A(t,p))$ tokens in each of its output places p. To allow a transition to fire, it is required that all input places contain the necessary amount of tokens for consumption, i.e., a transition is enabled if $\forall p \in A(p,t) : M(p) \geq W(A(p,t))$. Furthermore, it has to be ensured that the allowed capacity of target places is not exceeded after firing, i.e., $\forall p \in A(t,p) : W(A(t,p)) + M(p) \leq C(p)$. The firing of a transition changes the marking of the Petri Net in a way that the tokens of the according input places are deleted and the produced tokens are added to the marking of the output places, as can be seen in Fig. 6.1(b). Formally noted, firing a transition $t \in T$ changes a marking $M_1\colon P \to \mathbb{N}$ to marking $M_2\colon P \to \mathbb{N}$ in a way that $M_2 = M_1 - M_- + M_+$ whereby

$$M_-(p) = \begin{cases} W(A(p,t)), & p \in A(p,t), \\ 0, & \text{else.} \end{cases} \quad M_+(p) = \begin{cases} W(A(t,p)), & p \in A(t,p), \\ 0, & \text{else.} \end{cases}$$

Please note that in a Petri Net several transitions may be enabled at the same time, and if so, one transition may fire in a non-deterministic way. This is the main reason why Petri Nets are especially suited to model concurrent systems.

6.1.2 Colored Petri Nets in a Nutshell

Since CENs act primarily as theoretical models, many extensions have been proposed to make Petri Nets more flexible and applicable for practical use [72, p. 4]. These types of Petri Nets are often called *High-Level Petri Nets* and combine the functionality of Petri Nets with the functionality of programming languages. One prominent representant of High-Level Petri Nets are Colored Petri Nets (CPNs) [72]. The main idea of CPNs is to allow to attach data values to tokens, also called *token color*, in order to distinguish between different tokens. Every place is typed to a so-called *color-set* (data type) which determines the valid set of token colors (data values) (cf. Fig. 6.2(a)). To define color-sets, a so-called inscription language is provided, e.g., CPN ML[1], in case of the most prominent tool called CPN Tools[2]. CPN ML thereby bases on the functional programming language Standard ML [106, 154].

To extend the previous example from a standard Petri Net to a CPN, each place demands a color-set, e.g., the color-set `INT` in Fig. 6.2(a) and (b), meaning that tokens residing in these places require an integer value as their token color. To specify the initial marking an according inscription is required, e.g., in the example one token with the value 4 exists in the place `p1` and two tokens with value 4, one token with the value 5 and three tokens with value 6 exist in the place `p2`. The number of tokens is denoted by $x`token$, whereby x denotes the number of equal tokens. Furthermore, a marking of a place is a multiset, whereby each different set of tokens is delimited by $++$. As can be seen in this example, markings are defined as multisets, i.e., several tokens with the same color (value) are allowed. In order to query or produce tokens, CPNs allow

[1] http://www.daimi.au.dk/designCPN/man/Reference/Reference.Main3.CPN.ML.pdf
[2] http://cpntools.org

for variables (with optional capacity), or even functions on arcs (cf. `i+k` on arc from transition `t1` to place `p3` in Fig. 6.2(b)). The color-sets of the variables have to be equal to the color-sets of the according input or output place (cf. Fig. 6.2(a)). A transition is enabled, if every variable on arcs from an input place to a transition may be bound to an according token. Furthermore, transitions may specify a guard condition which needs to be fulfilled to enable a transition, i.e., the transition `t1` is only enabled if the variable `i` may be bound to a token whose value is greater than 3 in Fig. 6.2.

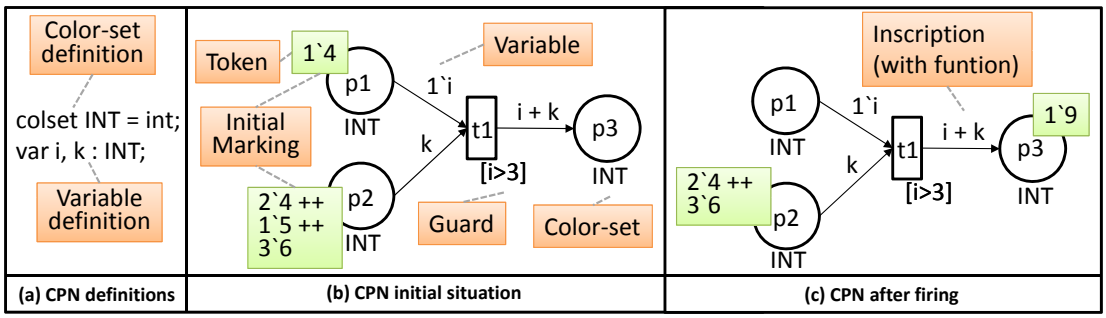


Figure 6.2: Simple Colored Petri Net

Formal Definition of Static Parts. In a formal manner, according to [72, p. 87], the structure of a CPN is defined as a nine-tuple $CPN = (P, T, A, \Sigma, V, C, G, E, I)$. P, T and A follow the same definitions as described above. Σ denotes a finite set of non-empty color-sets, i.e., the color-sets which are defined in a certain CPN, e.g., `INT` in our example. V is a finite set of typed variables such that Type$[v] \in \Sigma$ for all variables $v \in V$, i.e., every variable v needs to be typed to a defined color-set, e.g., `i` and `k` which are typed to the color-set `INT` in the example. $C\colon P \to \Sigma$ is a color set function that assigns a color set to each place, i.e., in the example in Fig. 6.2 C assigns the color-set `INT` to each place. $G\colon T \to Expr_v$ is a guard function that assigns a guard to each transition t such that $Type[G(t)] = Bool$, i.e., transitions may exhibit a boolean condition. $E\colon A \to Expr_v$ is an arc expression function that assigns an arc expression to each arc a such that $Type[E(a)] = C(p)_{MS}$, where p is the place connected to the arc a, i.e., the color-sets of the arc expression have to be equal to the color-set of the according place. $I\colon P \to Expr_\emptyset$ is an initialization function that assigns an initialization expression to each place p such that $Type[I(p)] = C(p)_{MS}$, i.e., a function to establish the initial marking of a CPN.

After explaining the structure of CPNs, in the following the matching and firing semantics of transitions in CPNs are described shortly, first informally by means of an example, followed by the formal definition thereof. The transition `t1` in Fig. 6.2(b) is enabled since there are valid *bindings* available that fulfill the guard. A binding denotes which value (token) is bound to which variable, i.e., in the example the variable `i` may only be bound to the value 4 of place `p1`. In contrast, the variable `k` may either be bound to the value 4, 5, or 6. Thus, there are several valid bindings available and one of them is chosen in a non-deterministic way, e.g., `i` has been bound to 4 and `k` has been bound to 5 in the example in Fig. 6.2(c), resulting in a target token with the value 9, since the outgoing arc of transition `t1` exhibits a simple function adding the two values.

Formal Definition of Dynamic Parts. The firing semantics of CPNs is formally defined in [72, p. 89]. Variables of a transition t are denoted by $Var(t) \subseteq V$ which might appear in guards and in arc expressions of arcs connected to t, e.g., for the above example $Var(t1)$ is defined as $\{i, k\}$. Furthermore, a binding of a transition t is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in Type[v]$. The set of all bindings for a transition t is then denoted by $B(t)$. In order to check if a binding enables a transition, the notion of binding elements is defined as a pair (t, b) such that $t \in T$ and $b \in B(t)$. To check enabling of a transition, first the guard has to evaluate to `true` for a binding element, denoted as $G(t)\langle b \rangle = true$. Second, all arc expressions on arcs to a transition t need to be fulfilled, i.e., $\forall p \in P : E(p,t)\langle b \rangle \ll= M(p)$. Thereby, the evaluation of $E(p,t)$ in the binding b specifies the multiset of tokens a transition t removes from a place p, which must be smaller or equal to the actual marking M of the place p.

6.1.3 Petri Net Markup Language

As may be seen by the previous discussion, various different types of Petri Nets are available. In order to allow interchange between different types of Petri Nets and their different tools used to model them, the Petri Net Markup Language (PNML) has been proposed as an XML-based interchange format for Petri Nets [161]. In this respect, the PNML is comparable to the role of metamodels in MDE. Since the PNML is used as a target metamodel for the compilation from Transformation Nets to CPNs, the most important parts are shortly explained in the following. In a first step, the common core concepts of Petri Nets are considered as can be seen in Fig. 6.3.

In PNML, a `PetriNetFile` forms the root container, which represents a file that might contain several `PetriNets`. Furthermore, each `PetriNet` consists of `Objects`, which basically model the graph structure of a Petri Net. Consequently, an object is either a `Place`, a `Transition` or an `Arc` (cf. according subclasses of `Object`). For modularization of Petri Nets, `Pages`, `RefPlaces` and `RefTransitions` are provided. `Pages` form sub-

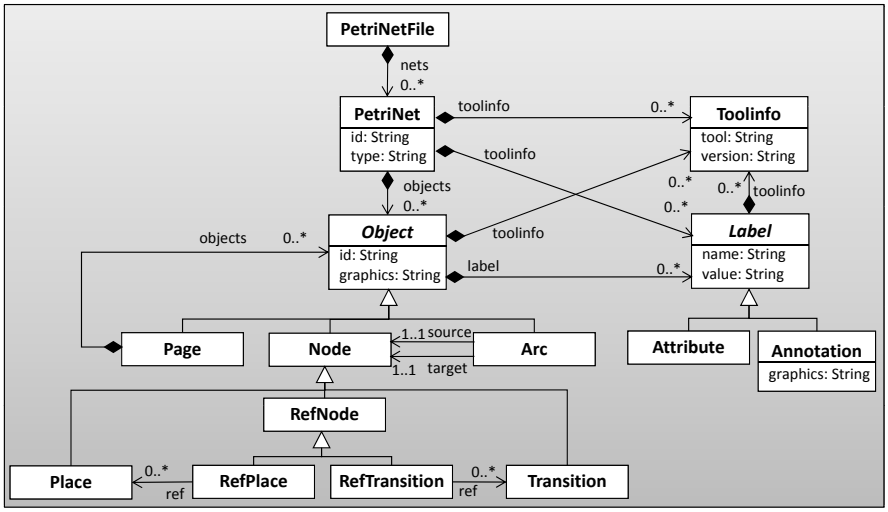


Figure 6.3: Core of Petri Net Markup Language [161]

nets since they are allowed to consist of further objects – even nested pages are allowed. In order to connect Petri Net `Nodes` on different pages, so-called `ReferenceNodes` are provided, i.e., a reference node may refer to any node of the Petri Net irrespective of the actual page. `Labels` are used to add further meaning to objects, ranging from the name to an initial marking or an arc inscription or even the definition of guards or functions in higher-level nets. Thereby, `Annotations` represent an infinite range of legal values, i.e., textual information as names or initial markings, whereas `Attributes` are restricted to a finite set of values (i.e., enumerations), typically influencing the graphical layout, i.e., an arc type `read` might result in a bidirectional arc. In order to store layout information, `Objects` and `Annotations` provide the attribute `graphics`. Finally, the class `ToolInfo` allows to store tool specific information. For further details, the reader is referred to [161].

In order to allow representing the concepts of high-level Petri Nets, e.g., CPNs as presented in Subsection 6.1.2, PNML allows to define so-called *Petri Net types*, which add definitions of labels or objects that are specific to a certain kind of Petri Nets. Such a Petri Net type for CPNs has been defined in [164], where e.g., specific labels are added to represent a marking of a place (e.g., `HLMarking`) or annotations (e.g., the class `Sort` to represent color-sets). The CPN-specific PNML metamodel is part of the ASAP framework [163] and is exactly the one used in the following to explain the compilation of Transformation Nets to CPNs. This metamodel assumes CPN ML as inscription language, which is used in the following as according inscription language as well.

6.2 Compilation of Static Parts of Transformation Nets

In a first step, the compilation of the static parts of Transformation Nets, i.e., places and tokens, is explained. Since the previous chapter introduced the static concepts of Transformation Nets in an informal manner only, the concepts of Transformation Nets are furthermore formalized in order to set the basis for the translation of Transformation Net concepts to equivalent concepts in standard Colored Petri Nets. Finally, the actual compilation is shown in detail.

6.2.1 Formalization of Static Parts of Transformation Nets

The static parts of Transformation Nets may be formally defined as a 5-tuple $StaticTNParts = (P, \Sigma_P, C, \Sigma_I, I)$, following the principles of standard CPNs. In this respect P denotes a finite set of `TNPlaces`. Σ_P is a finite set of predefined data types, i.e., $\Sigma_P = \{Class, Attribute, Reference, TracePlace\}$. $C\colon P \rightarrow \Sigma_P$ is a data type function that assigns a data type to each place. This means that every place has to be typed to either `Class`, `Attribute`, `Reference` or `TracePlace` in Transformation Nets. Σ_I is a finite set of predefined token types, i.e., $\Sigma_I = \{Object, Value, Link, Trace\}$. Finally, I is an initialization function that assigns a token t to a place p such that

$$Type[I(t)] = \begin{cases} \textit{Object}, & \textit{if Type[p]} = \textit{Class} \\ \textit{Value}, & \textit{if Type[p]} = \textit{Attribute} \\ \textit{Link}, & \textit{if Type[p]} = \textit{Reference} \\ \textit{Trace}, & \textit{if Type[p]} = \textit{TracePlace} \end{cases}$$

As may be seen from these definitions, the static parts of Transformation Nets correspond to the according definitions in CPNs. Since Transformation Nets provide a DSL on top of CPNs, complexities are hidden. This is achieved by restricting the set Σ of CPNs to a fixed, predefined set of datatypes in Transformation Nets, denoted by Σ_P. This simplification also leads to a simplification concerning tokens. Since tokens (data values) have to conform to their color-set (data types) of places, the set of valid tokens may also be restricted to these four different types (cf. Σ_I). In the following, it is shown that the static parts of Transformation Nets may be compiled to according concepts of CPNs.

6.2.2 Compilation of Metamodels and Models

In order to compile Transformation Net places to according places in CPNs, first color-sets need to be defined. Since the possible types of places in Transformation Nets are limited to exactly four, according color-set definitions expressed in terms of CPN ML may be automatically generated. Therefore, the following definitions result:

$$\begin{aligned} \textit{colset Class} &= \textit{record oid: STRING * t: STRING;} \\ \textit{colset Attribute} &= \textit{record obj: Class * vid: STRING * v: STRING;} \\ \textit{colset Reference} &= \textit{record source: Class * target: Class;} \\ \textit{colset SourceCtx} &= \textit{list Class;} \\ \textit{colset TracePlace} &= \textit{record source: SourceCtx * target: Class;} \end{aligned}$$

The definition of color sets is aligned to the metamodel of Transformation Nets. The color-set $Class$ defines a record, which consists of an `oid` component to identify according objects as well as the component `t` to store the type of the according object. The color-set $Attribute$ again defines a record which consists of a component `obj` which identifies the object the according value belongs to. In order to store the actual value, again a unique id is derived for every value (cf. component `vid`) and the actual value is stored as string in the component `v` of the record. This means that if the actual value needs to be accessed (in a condition or function), according casts are required, e.g., the string "1" may be casted to an integer value 1 by the function `valOf(Integer.fromString())`. Alternatively, it would have been also possible to specify a specific `Attribute` colorset for every primitive datatype. This was omitted in order to keep the actual transformation logic specified independent of the concrete datatype, e.g., in case of modules, according `AttributePorts` can be bound to any attribute, only the arc inscriptions need to be accordingly updated (cf. below). The record for the color-set $Reference$ consists of a `source` and a `target` component which refer to the source and target objects of

a link[3]. The definition of the color-set $TracePlace$ first requires the definition of the color-set $SourceCtx$ as a list of $Class$. This list is used in the record definition of $TracePlace$ to store which source objects have been used (cf. component $source$ of record) for the production of an object (cf. component $target$ of the record).

After defining the required color-sets, places in Transformation Nets may be compiled to according CPN places, i.e., places derived from the source and target metamodel as well as trace and intermediate places have to be translated into according places in CPNs. By this, every instance of a *TNPlace* in Transformation Nets is compiled into a `Place` instance in PNML (cf. corresponding numbers in Fig. 6.4). Additionally, in order to assign a name to the place, the value of the attribute `TNPlace.name` is compiled into an according `Name` instance, having the attribute `Name.text` set to the value of the attribute `TNPlace.name`. As an example, the Transformation Net places named `Package`, `classes`, `Class` and `isPersistent` in Fig. 6.4 (a) (concrete syntax) and (b) (abstract syntax), are compiled into according `Place` and `Name` instances (cf. Fig. 6.4(c) and (d)). Finally, based on the actual type of a `TNPlace`, i.e., either `Class`, `Attribute`, `Reference` or `TracePlace`, the color-set of the place in CPNs has to be derived. The PNML metamodel represents color-sets by means of instances of the class `Sort` whose attribute `Sort.text` must be equal to a defined color-set. For example, if a place of type `Class` in Transformation Net is compiled to CPNs, a `Sort` instance is produced whose attribute `Sort.text` is set to `"Class"`, which corresponds to the name of the above defined color-set (cf. Fig. 6.4).

After the compilation of places, the contained tokens have to be translated into according tokens in CPNs. The Transformation Net DSL hides the complex specification of tokens in CPNs by means of CPN ML inscriptions. In Transformation Nets a predefined set of tokens exists, i.e.,

[3]Source and target are typed to class in order to explicate that references are between classes although for the implementation the according ids would suffice. The same is ture for the definitions of the Attribute and TracePlace color sets.

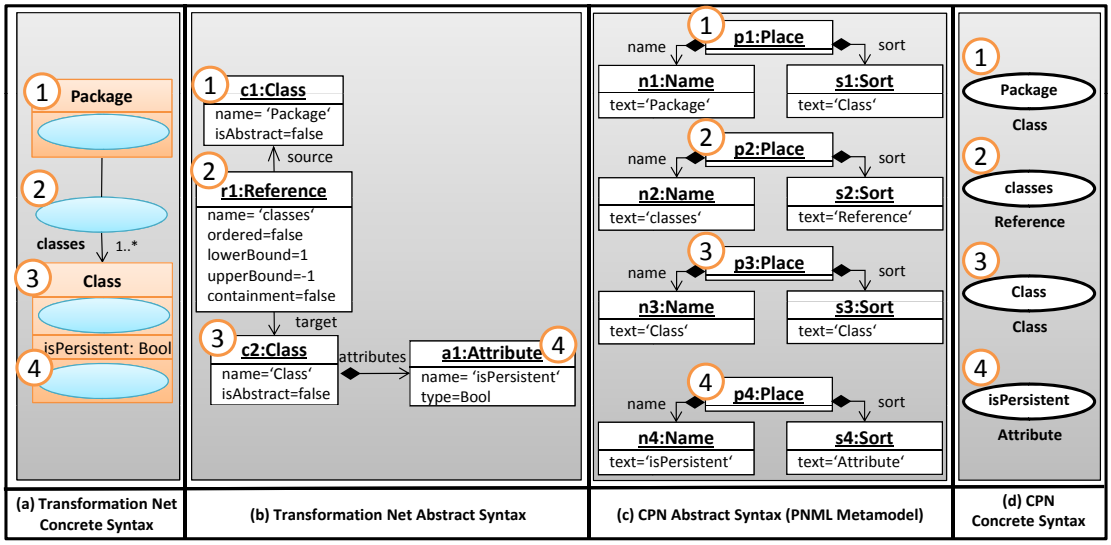


Figure 6.4: Compilation of Transformation Net Places to CPNs

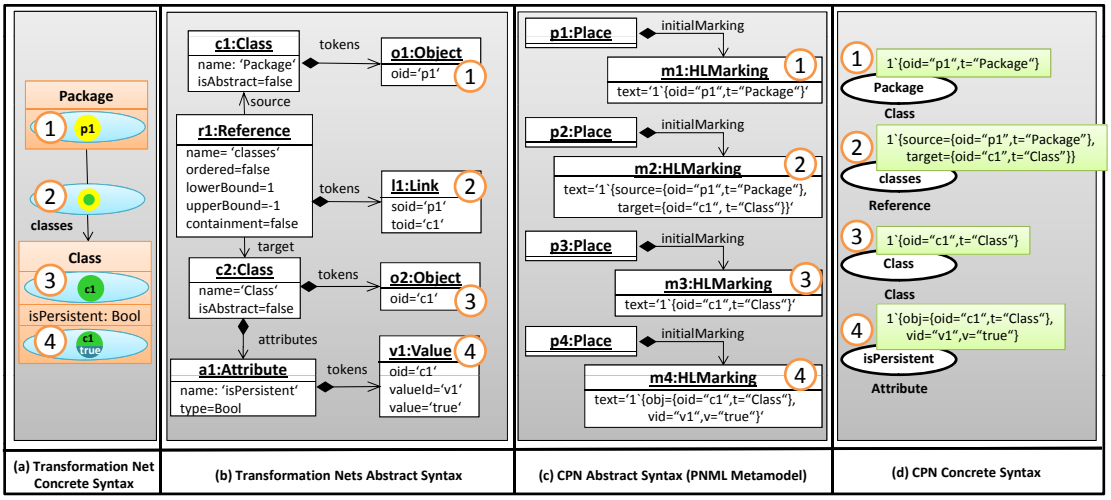


Figure 6.5: Compilation of Transformation Net Tokens to CPNs

`Object`, `Value`, `Link`, and `Trace` (cf. Σ_I in the formal definition), which may only reside in type compatible places (cf. definition of I). As can be seen in Fig. 6.5, every token is compiled to an instance of the class `HLMarking`, which is used to represent a marking (tokens) in CPNs. Depending on the type of the token in Transformation Nets, a different inscription has to be derived in a way that the inscription corresponds to the defined color-set. For example, for an `Object` token, the `oid` component of the `Class` color-set record is set to the attribute `Object.oid` and the type component `t` to the value of the attribute `Class.name` of the according `Class` place. The same principle is followed for `Value` and `Link` tokens.

Inheritance. Since metamodels incorporate the concept of inheritance, it has to be represented by appropriate CPN concepts as well. Unfortunately, CPNs per se do not support inheritance between color-sets. Nevertheless, as will be described in Section 6.4 in detail, it should be possible that a transition which matches for tokens of supertypes also matches for tokens of subtypes. Therefore, tokens in the places of subtypes should be copied to the corresponding place of the supertype during compilation. This scenario is depicted in Fig. 6.6 whereby the class `Y` inherits from the class `X`. To allow CPN transitions to match for a token of a subclass, e.g., token `y1`, the marking is copied to the marking of the according supertype, e.g., marking of place `X` now contains a marking comprising tokens `x1` and `y1`, meaning that tokens are duplicated.

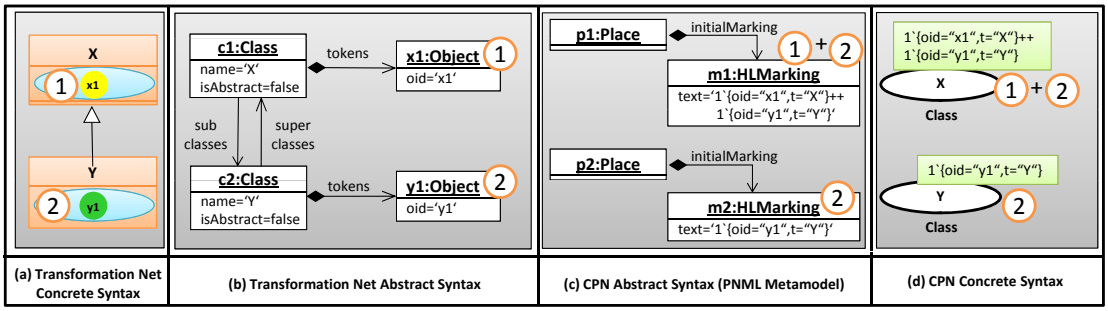


Figure 6.6: Compilation of Inheritance Relationships in Transformation Nets to CPNs

6.3 Compilation of Dynamic Parts of Transformation Nets

After describing the compilation of the static parts, the focus is shifted to the dynamic parts of Transformation Nets and its compilation to CPNs in the following. For this, again the concepts of Transformation Nets are formalized, followed by a detailed description of the according compilation.

6.3.1 Formalization of Dynamic Parts of Transformation Nets

In order to incorporate the dynamic aspects into the formal definition, the $StaticTNParts = (P, \Sigma_P, C, \Sigma_I, I)$ must be extended to a 13-tuple $TransformationNet = (P, \Sigma_P, C, \Sigma_I, I, T, PT, \Sigma_{Pt}, PTF, PTT, A, G, E)$. Thereby, T denotes a finite set of transitions such that $P \cap T = \emptyset$, which is similar to standard CPNs. PT defines a finite set of `Patterns`. Since in Transformation Nets it is only necessary to match for either objects, attributes, references or trace tokens, Σ_{Pt} defines a set of predefined types of patterns, i.e., $\Sigma_{Pt} = \{ObjectPattern, Value - Pattern, LinkPattern, TracePattern\}$, used to hide inscriptions from the transformation designer. $PTF\colon PT \rightarrow \Sigma_{Pt}$ is a function that assigns a pattern type to each pattern, $PTT\colon PT \rightarrow T$ is a function that assigns a non-empty set of patterns to each transition. Furthermore, the set of arcs A is a subset of the cross product of places P and patterns PT, i.e., $A \subseteq P \times PT \cup PT \times P$ such that $\forall a, b \in A : a(p, pt) \Rightarrow \nexists b(pt, p))$ and $\forall a, b \in A : a(pt, p) \Rightarrow \nexists b(p, pt))$ where $pt \in PT, p \in P$, i.e., a pattern is either a query pattern or a production pattern, but not both. Furthermore, the types of places and patterns have to be compatible, i.e, function b has to evaluate to `true`, such that:

$$b = \begin{cases} true, & \textit{if Type[p] = Class} \wedge \textit{Type[pt] = ObjectPattern} \\ true, & \textit{if Type[p] = Attribute} \wedge \textit{Type[pt] = ValuePattern} \\ true, & \textit{if Type[p] = Reference} \wedge \textit{Type[pt] = LinkPattern} \\ true, & \textit{if Type[p] = TracePlace} \wedge \textit{Type[pt] = TracePattern} \\ false, & \textit{else}. \end{cases}$$

This is different, compared to the definition in standard CPNs, since in CPNs arcs are directly connected to transitions, whereas in Transformation Nets there is a further step of indirection via patterns. This further step of indirection allows to derive the complex arc inscription automatically (cf. below). $G\colon T \rightarrow Expr_v$ is a condition function that assigns a condition to each transition t such that Type[$G(t)$]$= Bool$, i.e., transitions may exhibit a boolean condition, equal to standard CPNs. Finally, $E\colon A(pt, p) \rightarrow Expr$ is an arc expression function that assigns an arc expression to arcs a which originate from a pattern pt and target a place p such that Type[$E(a(pt, p)$]=C(p)$_{MS}$, i.e., only outgoing arcs of transitions might exhibit functions in Transformation Nets.

In summary, the main difference between Transformation Nets and standard CPNs lies in the introduction of patterns and explicit pattern types. The introduction of patterns in the Transformation Net DSL is, nevertheless, convenient for simulation, since it may easily be followed which tokens are bound to which patterns.

The firing semantics follows the same principles as CPNs, but needs to handle the semantics of patterns. Patterns exhibit a predefined set of variables, which need to be bound during matching, denoted by $Var(pt)$ such that

$$Var(pt) = \begin{cases} \textit{oid, typeName}, & \textit{if Type[pt] = ObjectPattern} \\ \textit{oid, typeName, valueId, value}, & \textit{if Type[pt] = ValuePattern} \\ \textit{soid, sType, toid, tType}, & \textit{if Type[pt] = LinkPattern} \\ \textit{sctx, target}, & \textit{if Type[pt] = TracePattern}. \end{cases}$$

Please note that the concrete names of the pattern variables are user-defined (cf. according attributes in the pattern classes in the metamodel in Fig. 4.3 on page 73) whereas the names above are of symbolic nature, only. Nevertheless, the number of variables per pattern is fixed, e.g., an `ObjectPattern` always has two variables that need to be bound during matching. Variables of a transition t are denoted by $Var(t)$ being the set of all pattern variables of a certain transition. Consequently, this leads to an equal firing semantics as stated above, i.e., the desired semantics of Transformation Nets may be expressed in terms of CPN concepts. The following subsection describes the compilation in detail.

6.3.2 Compilation of Transformation Logic

In order to exemplify the compilation of transformation logic, Fig. 6.7(a) shows the Transformation Net already presented in Fig. 4.17 on page 89. Thereby, `Packages` should be transformed into according `Schemas` and `Classes` to `Tables`, but only if the according Schema has been translated before. To express this dependency, a trace place between the two transitions is used. Fig. 6.7(b) shows the result of the compilation in CPNs, which is explained in detail in the following, focusing on the dynamic parts of Transformation Nets. Fig. 6.8 shows the details thereof by means of the abstract syntax.

Transition. In a first step, `Transitions` in Transformation Nets are compiled into `Transitions` in CPNs. Thus, in our example, two instances of transitions occur, whereby the name of the Transformation Net transition is simply copied (resulting in `Name` instances in the abstract syntax), depicted by ⑤ and ⑩ in Fig. 6.7 and Fig. 6.8.

Patterns. In a second step, arcs as well as query and production patterns of Transformation Nets have to be compiled. Whereas in Transformation Nets, patterns are source or target of arcs, in CPNs transitions are source or target of arcs. Therefore, the source or target of a CPN arc has to refer to the corresponding CPN transition, which originated from a Transformation Net transition the pattern is contained in. As an example, the arc `arc1` (cf. ⑥ in Fig. 6.8(a)) targets the object pattern `p1`, which is again contained in transition `t1` (cf. ⑤). Since the transition `t1` in the Transformation Net was compiled to transition `t1` in the CPN, the generated arc `a1` targets the transition `t1` (cf. ⑥ in Fig. 6.8(b)). In order to be able to match for tokens in CPNs, arcs require according arc inscriptions which are derived from the patterns in Transformation Nets. As stated before, every type of pattern exhibits a predefined number of variables which have to be correspondent with the defined color-set. Thus, for the different types of patterns different variables need to be derived, as described in detail in the following.

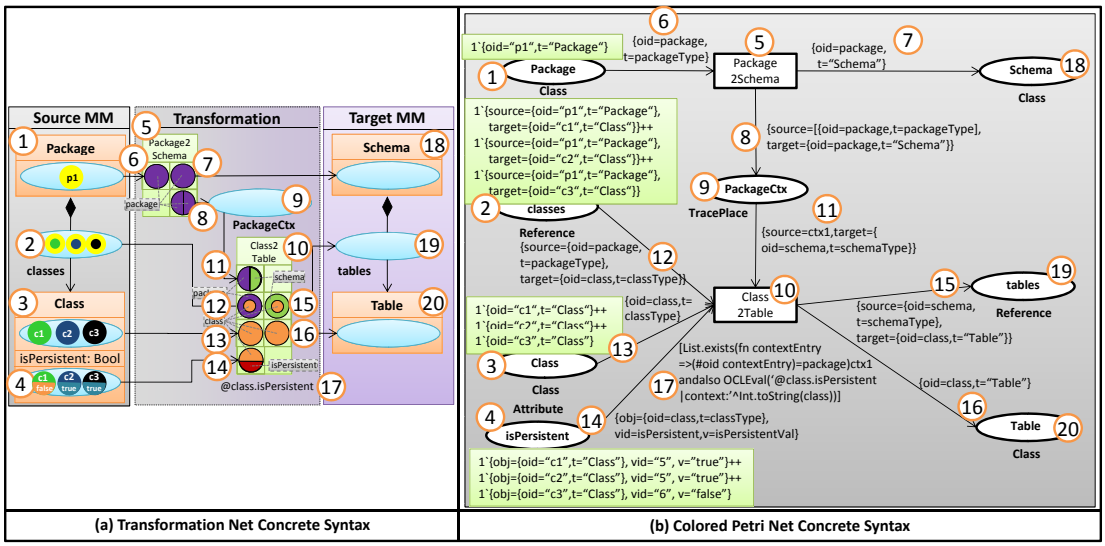


Figure 6.7: Compilation of Transformation Nets to CPNs in Concrete Syntax

- *ObjectPattern:* The record color-set `Class` offers the components `oid` and `t`. Consequently, in order to be able to match for object tokens, according variables need to be derived. For the component `oid` the according variable specified in the pattern (cf. attribute `ObjectPattern.oidVar`) is taken, e.g., `package` in case of `a1` (cf. ⑥ in Fig. 6.8). For the second component, again the variable of the patten is taken, but concatenated with the postfix `'Type'`. These variables are then used in the according arc inscription, e.g., for `a1` the arc inscription `{oid=package,t=packageType}` is derived (cf. ⑥ in Fig. 6.7(b) and Fig. 6.8(b)). For an arc from a transition to an output place, the variable `t` is set to the value of the according class, i.e., "Schema" for the arc `a6`

- *ValuePattern:* The record color-set *Attribute* consists of the components `obj` referring to the object (which again consists of the components `oid` and `t`) and `vid` and `v` for the actual attribute value. `ValuePatterns` offer two variables, i.e., `ValuePattern.oidVar` and `ValuePattern.valueIdVar` which are used to derive the according arc inscription. Thereby, `ValuePattern.oidVar` is used to derive the inscription for the according object as described before. Additionally, the value of the attribute `ValuePattern.valueIdVar` is used as a variable for the component `vid` and as a variable for the component `v` of the record. For this, the value of the attribute `ValuePattern.valueIdVar` is concatenated with the postfix `'Val'`. To exemplify this ⑭ in Fig. 6.8 shows the resulting arc inscription, e.g., since the `ValuePattern.oidVar` of `p7` is set to `'class'` and `ValuePattern.valueIdVar` to `'isPersistent'` the corresponding arc inscription is `{obj={oid=class, t=classType}, vid=isPersistent, v=isPersistentVal}`.

- *LinkPattern:* The record color-set `Reference` consists of the components `source` and `target` which identify a certain source and target object. In this respect, the deriva-

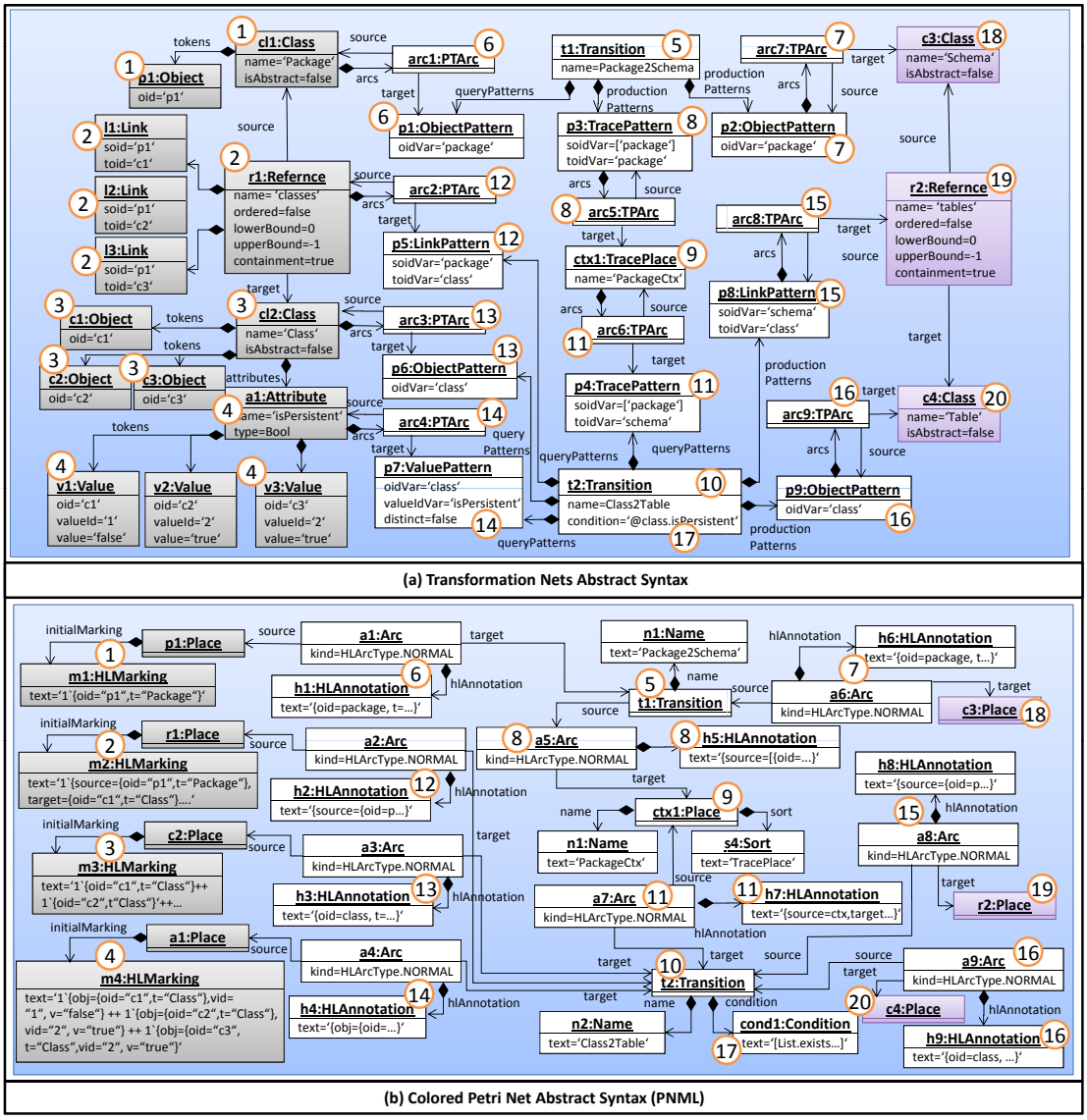


Figure 6.8: Compilation of Transformation Nets to CPNs in Abstract Syntax

tion of the arc inscription is equal to `ObjectPatterns`. As can be seen in (12) in Fig. 6.8 and in Fig. 6.7, the value `LinkPattern.soid` of `p5` is set to `'package'` and `LinkPattern.toid` to `'class'`, resulting in the arc inscription `{source={oid= package, t=packageType}, target={oid=class,t=classType}}`.

- *TracePattern:* `TracePatterns` indicate which source objects are used to generate a certain target object. Since source objects might be merged, the record color-set `Trace-Place` contains a `source` component which is itself a list of `Class` denoting the ac-

cording objects. Nevertheless, only one target element is allowed and therefore the component `target` is typed to a single `Class`, only. If a *production trace pattern* is considered, the arc inscription may be created in a similar manner as described before, e.g., as can be seen in ⑧ in Fig. 6.8, the array `TracePattern.soidVar` contains the variable `'package'`. The elements of the array are used to derive the arc inscription of the source component, whereas the `TracePattern.toidVar` is used to derive the target component (in an equal way as done for object patterns). Thus, in our example the derived arc inscription is `{source= [{oid=package, typeName=packageType}], target={oid=package, typeName=packageType}}`. Please note that since the objects are only copied, source and target components of the trace pattern are equal, which needs not always be the case. Additionally, if several source objects are merged, the source component list contains several entries, i.e., all the objects that should be merged.

In case that trace information should be queried by a transition, the arc inscription needs to be adapted accordingly. Instead of producing new trace information, the arc inscription, together with an according guard expression, needs to query, if a certain trace token contains a certain source object. In ⑪ in Fig. 6.7(b) the arc inscription `{source = ctx1, target={oid=schema,t=schemaType}}` queries the context token. Thereby, the variable `ctx1` is automatically generated, whereby the string `ctx` is concatenated with the index of the context query token, whereas the arc inscription for the `target` component is derived from the attribute `TracePattern.toidVar`. The specified guard `List.exists(fn contextEntry ⇒ (#oid contextEntry)= package) ctx1` checks if the component `source`, which is bound to the variable `ctx1` in the arc inscription, contains an `oid` which is equal to the value bound to the variable `package`, i.e., if a certain `Package` was transformed to a certain `Schema`.

As the attentive reader might have already spotted, the execution of the compiled CPN depicted in Fig. 6.7(b) does not produce the expected tokens (as shown in Fig. 4.17). This is since the transition `Class2Table` might only fire once, as the only available trace token is consumed by the first firing of the transition. Consequently, the transition `Class2Table` is not enabled twice, resulting in either the net presented in Fig. 6.9(a) or the net in Fig. 6.9(b), since only one of the persistent classes is transformed to an according table. As described in Subsection 4.4.1, transitions in Transformation Nets do not consume tokens but just read the tokens of the connected input places in order to avoid erroneous race conditions, e.g., that might occur in case of 1:n references as indicated in Fig. 6.9. Therefore, the changed default consumption behavior has to be accordingly considered in the compilation process as described in the following.

Non-consuming Firing Behavior. In order to realize the non-consuming firing behavior in CPNs, three adoptions are needed. First, arcs from places to transitions need to be changed to so-called *test arcs* to read tokens from input places only instead of consuming them. Second, a so-called *history place* to track the combinations of tokens already fired needs to be introduced. Finally, the *guard condition* needs to be accordingly adapted to prevent an infinite number of firings. Considering the first point, the attribute value `Arc.kind` needs to be changed from `HLArcType.NORMAL` to `HLArcType.TEST` which results in a bidirectional arc, i.e., the tokens that are only read from the input place (cf. e.g., ⑥ in Fig. 6.10). Nevertheless, this would

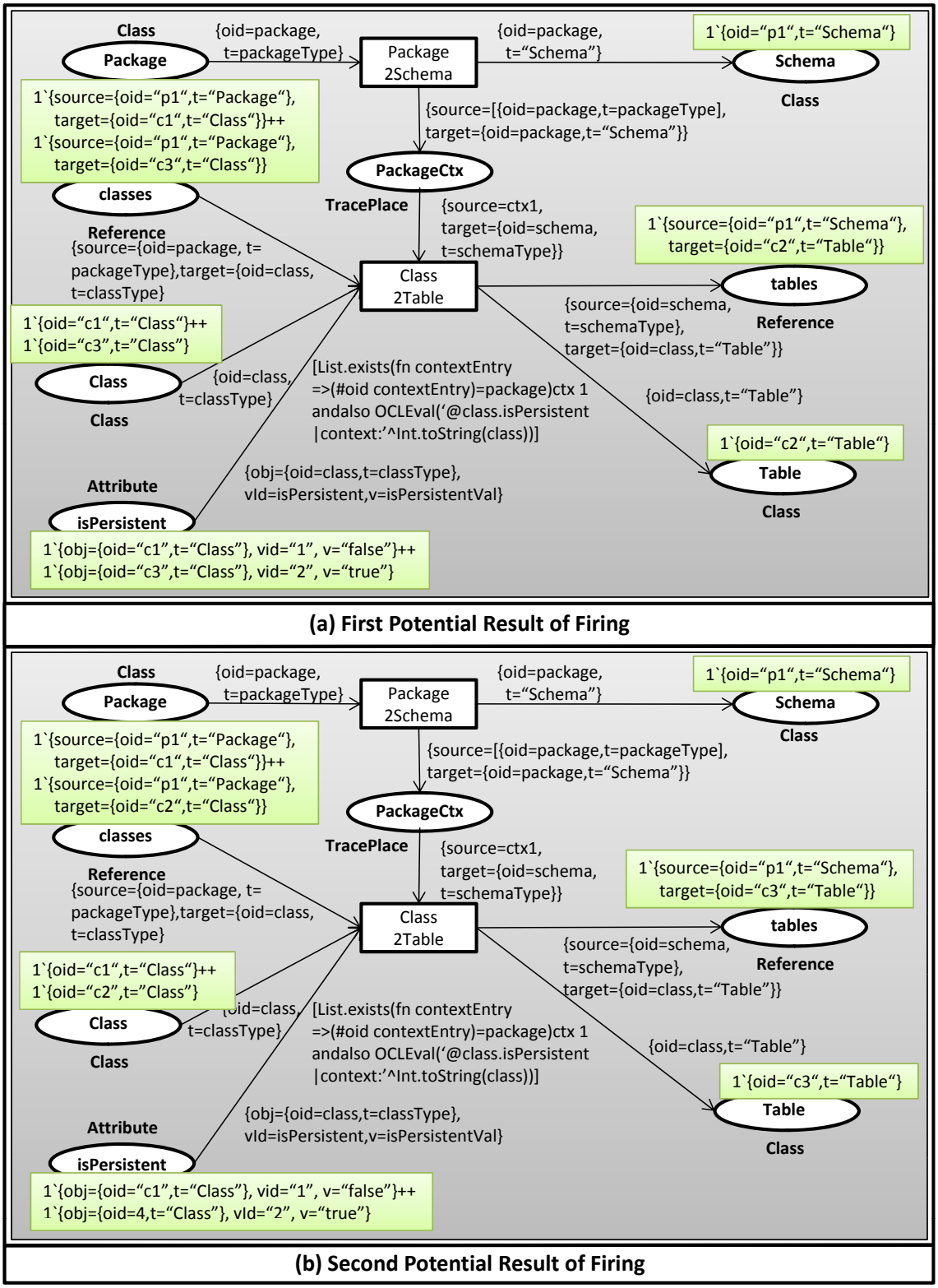


Figure 6.9: Erroneous Consumption of Source Tokens

lead to an infinite sequence of firings, since these tokens could be matched over and over again. In order to prevent this, the concept of a history place is introduced, which is defined by means of the following color-set definition, representing a list that stores lists of strings, which represent the ids of the involved elements. The `History` color-set is defined as follows:

$$\textit{colset IDs} = \textit{list STRING;}$$
$$\textit{colset History} = \textit{list:IDs;}.$$

This history list is then matched by the according transition (cf. variable `hist` in Fig. 6.10). After firing, the list is put back into the history place, whereby the matched token configuration is inserted in a sorted manner by an according arc inscription which makes use of the custom function `InsertSorted`, shown in Listing 6.1. The sorting is important in order to achieve a unique state space, as detailed in Section 7.4. The function `InsertSorted` first checks if the list to insert is nil – then it returns the current history – or the history is nil – then it wraps the list into another list (since history is a list of lists) and returns it. Otherwise it is checked if the list to insert is smaller than the first list of the history (cf. variable `h` – line 3) by means of the custom function `ListSmaller`, which compares two lists. If the list is smaller than the list of the history entry, it may be prepended, otherwise the function to find the correct insertion position is called recursively.

Listing 6.1: Functions for Inserting History Tokens in a Sorted Manner

```
1  fun InsertSorted nil history = history (*empty list*)
2      | InsertSorted l nil = [l] (*empty history*)
3      | InsertSorted l (h::history) =
4          if ListSmaller(l,h) then (*prepend smaller list*)
5              l::h::history
6          else
7              h::InsertSorted l history (*recursive call*)
8  ------------------------------------------------------------
9  fun ListSmaller(nil:IDs, list2:IDs) = true (*first list is empty*)
10     | ListSmaller(list1:IDs, nil:IDs) = true (*second list is empty*)
11     | ListSmaller (x::list1:IDs) (y::list2:IDs) =
12         if x < y then (*found a smaller entry*)
13             true
14         else if x=y then
15             ListSmaller(list1,list2) (*recursive call*)
16         else
17             false
```

Nevertheless, the history is only able to prohibit multiple firings for equal token configurations in combination with an according guard condition. As can be seen in Fig. 6.10, the guard checks if the history already contains a list that is similar to the matched token configuration. For this, the custom function `Contains` is provided which compares two lists, whereby empty string values are not considered for comparison (needed in case of rule inheritance as discussed in the following), as shown in Listing 6.2. If the history already contains the matched token configuration, the guard fails and prohibits another firing for the same token configuration. As can be seen in Fig. 6.10(b), the according CPN is now able to match for both classes since the trace token is not consumed, resulting in the expected target model.

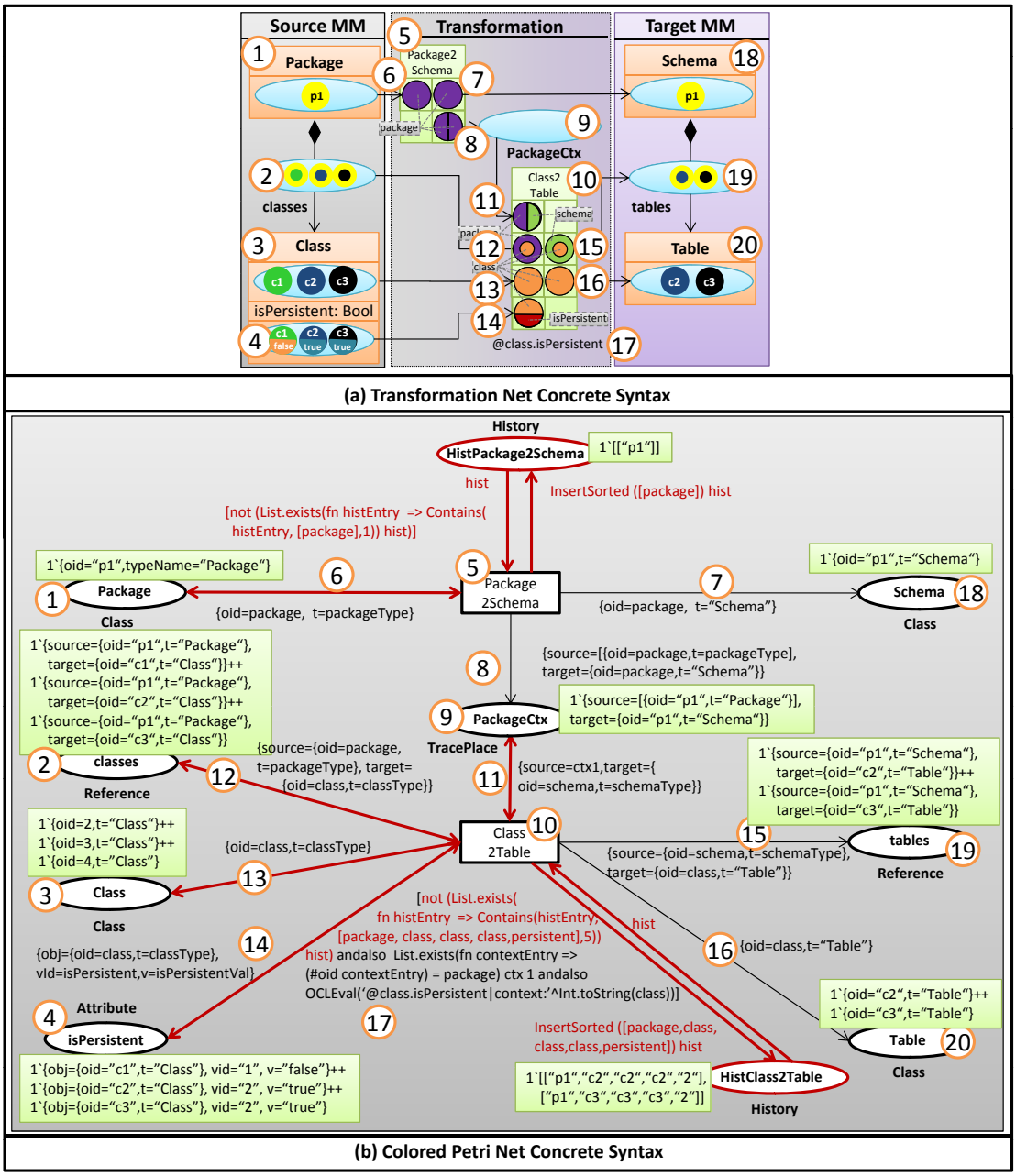


Figure 6.10: Compilation of Non-Consuming Firing Behavior

Listing 6.2: Function to Check if two Lists are Equal

```
1 fun Contains(curList,tokens,length)=
2     if curList=nil then false (*empty list*)
3     (*not equal length*)
4     else if (not (List.length curList = List.length tokens)) then false
```

```
5     else if length > 0 andalso (*compare entry*)
6       List.nth(curList,length-1) = List.nth(tokens,length-1) then
7         Contains(curList,tokens,length-1)
8     (*ignore empty string*)
9     else if length > 0 andalso List.nth(curList,length-1) = "" then
10        Contains(curList,tokens,length-1)
11    (*ignore empty string*)
12    else if length > 0 andalso List.nth(tokens,length-1) = "" then
13        Contains(curList,tokens,length-1)
14    else if length <= 0 then true (*compared all values*)
15    else false
```

Negative Patterns. In CPNs, the matching process assumes the existence of a certain token, e.g., if a class token is available generate a table token. Consequently, matching of non-existing tokens requires more complex structures. It is necessary to provide a list of tokens to check if a certain token is not contained, following the inhibitor arc pattern presented in [111]. In order to compile negative patterns, first only the tokens contained in an according input place, i.e., the places which are connected to a negated pattern, are wrapped into a list, for which the following color-sets are defined:

$$\begin{aligned} \textit{colset ClassList} &= \textit{list Class;} \\ \textit{colset AttributeList} &= \textit{list Attribute;} \\ \textit{colset ReferenceList} &= \textit{list Reference;} \\ \textit{colset TraceList} &= \textit{list TracePlace;} \end{aligned}$$

Since the reference pattern is negated in the example shown in Fig. 6.11, the tokens of place `superClass` typed to the color-set `Reference` are replicated in an according list which is put into the place `ListSuperClasses` typed to the color-set `ReferenceList`. The transition then reads the list of tokens instead of a single token. The guard condition of the transition checks if the list does not contain the negated token, e.g., in our example the guard checks if the value of variable `class` does not occur as a source of a reference (cf. expression `(#soid negatedElem)` which delivers the value of the `soid` component of a reference). As only the class `c1` does not have any superclass, i.e., it is a root class, only this class is transformed to an according table.

Distinct Values and New Colors. Often only for distinct attribute values a certain target element should be generated, as discussed in Subsection 4.4.1. In order to decide whether a target element should be generated for a certain source value, the already processed values need to be stored in a list (cf. ④ in Fig. 6.12). For this, the color-set `DistinctList` is provided, which is defined as follows:

$$\begin{aligned} \textit{colset IDs} &= \textit{list STRING;} \\ \textit{colset DistinctList} &= \textit{record value:IDs * target:Class.} \end{aligned}$$

The component `value` of the color-set `DistinctList` is defined as a list of strings in order to store combinations of distinct values, since it is allowed that a single transition in Transformation Nets exhibits several distinct patterns. In order to provide trace information, i.e., if the

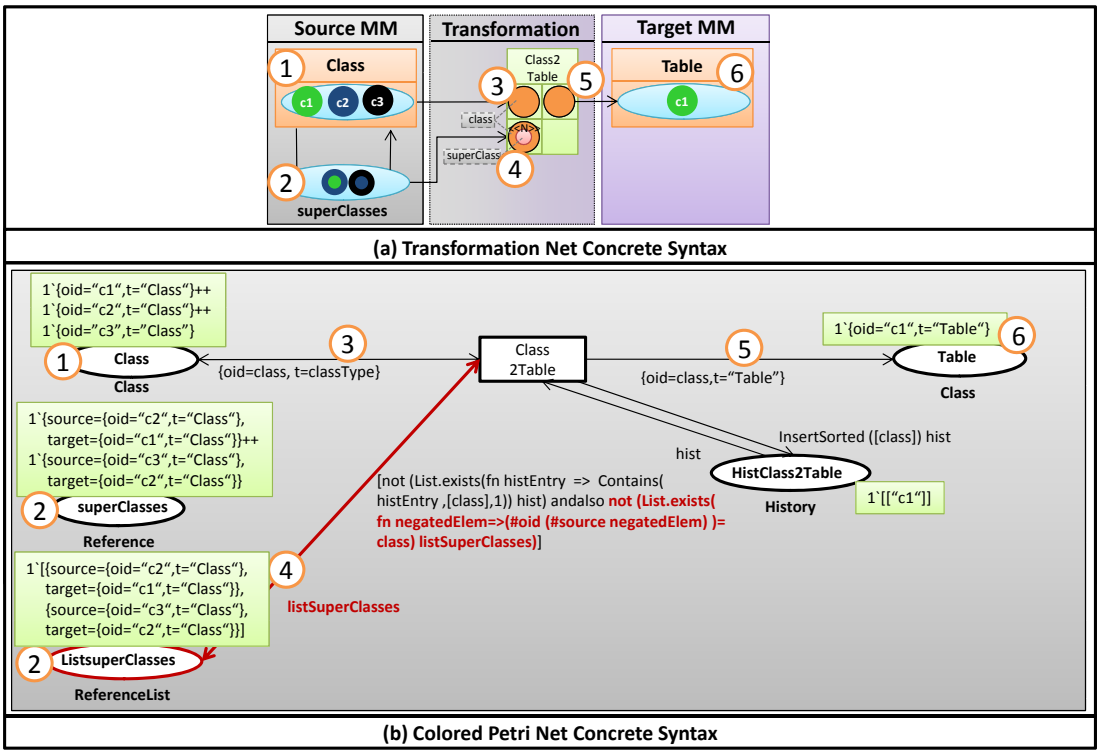


Figure 6.11: Compilation of Negative Pattern

according object the attribute belongs to is used in a production trace pattern, the list additionally stores which value was translated to which object by means of the component `target`. As can be seen in ⑤ in Fig. 6.12, the arc from the transition to the place creates a new token only if the list `values` did not contain the matched value before. Please note that no guard condition was used to prevent firing, i.e., the transition should not be enabled if the list contains a certain combination of values, since trace information should be provided for all possible combinations of values. In this respect, the complex arc inscription labeled ⑥ in Fig. 6.12 first checks if an entry in the list `values` is available for the combinations of the matched values. If this is the case, then the according value of the `target` component of the list entry is selected and set as value of the `target` component of the to be created trace token. Otherwise, a new trace token is created. Therefore, the trace place `Ctx` contains three tokens whereas the list of the place `typeVals` solely contains two tokens. The Transformation Net uses a new, unbound color to represent the newly generated `Type` objects, as can be seen in ⑤ in Fig. 6.12, meaning that a new id (color) needs to be generated for every produced token. In order to provide a new id, a global variable `newColor` is defined. Please note that if a global variable is used, it needs to be prefixed by an to receive the actual value, as may be seen for example in the arc inscription ⑤ in Fig. 6.12. The variable `newColor` is initialized by the highest id available, i.e., 5 in the example, and is incremented after firing by means of the function `increment`, in the action

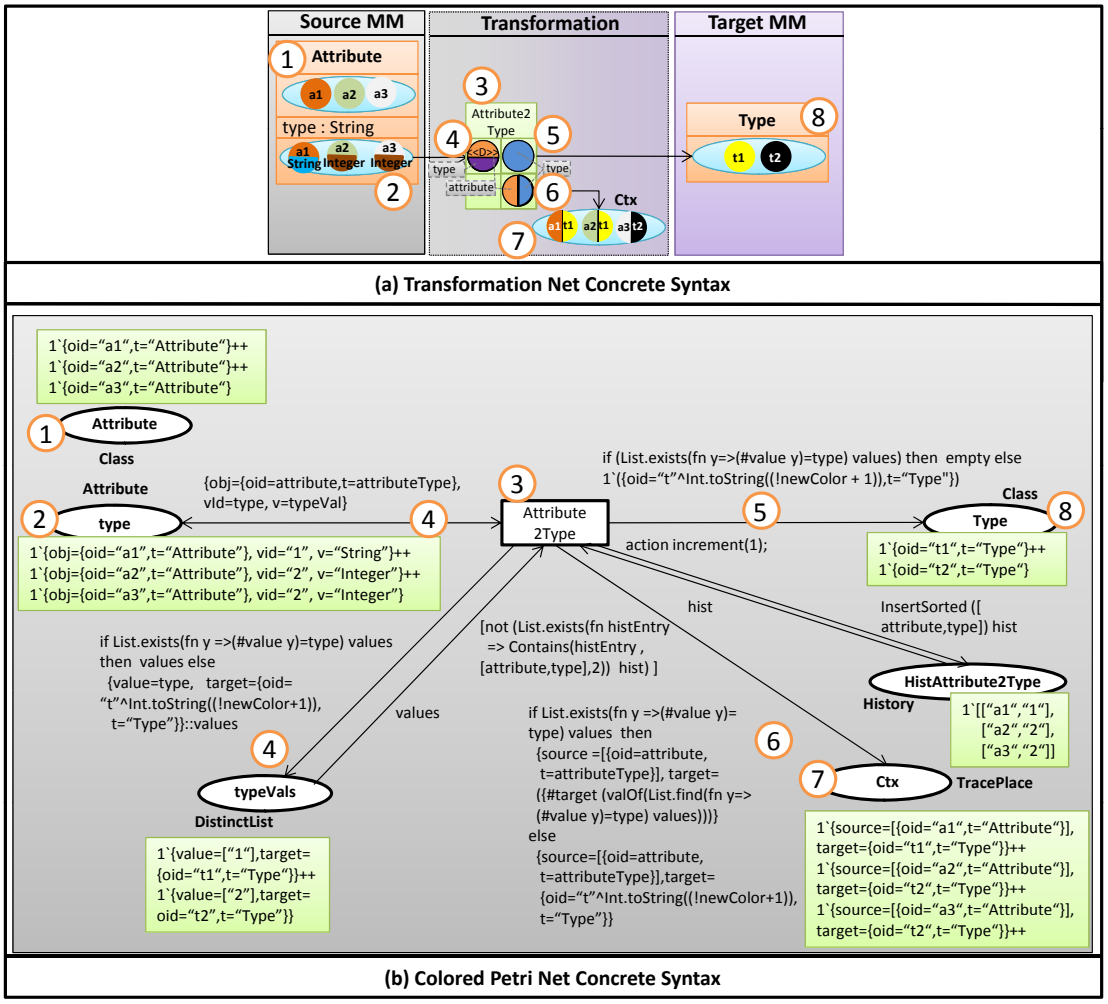


Figure 6.12: Compilation of Distinct Values and New Colors

block of a transitions. Action blocks of CPN transitions allow to specify code that should be executed immediately after a transition fired. Therefore, a new id is generated every time the transition fires.

Check Before Enforce. Finally, in order to omit the creation of duplicate elements on the target side, i.e., check before enforce semantics, the already produced target elements have to be stored in a list. For this the color-set `CBEVals` is provided, which stores a list that contains lists of ids, defined as follows:

$$colset\ IDs = list\ STRING;$$
$$colset\ CBEVals = list\ IDs;.$$

In the example depicted in Fig. 6.13 the production pattern exhibiting the variable `package` is marked as check before enforce. Consequently, the compiled CPN contains an additional place

CBEClass2Table which stores the ids of the already created Schema instances. This means that if the transitions fires the first time, no entry is contained in the place CBEClass2Table and therefore the arc inscription on the arc to the Schema output place (cf. (10) in Fig. 6.13) produces the according Schema instance. The according id of the generated token is put into the CBEClass2Table place, e.g., p1 in the example. When the transition is enabled a second time, again the token package with the p1 is bound. Nevertheless, since the place CBEClass2Table now already contains the id (cf. Contains function in (10) in Fig. 6.13),

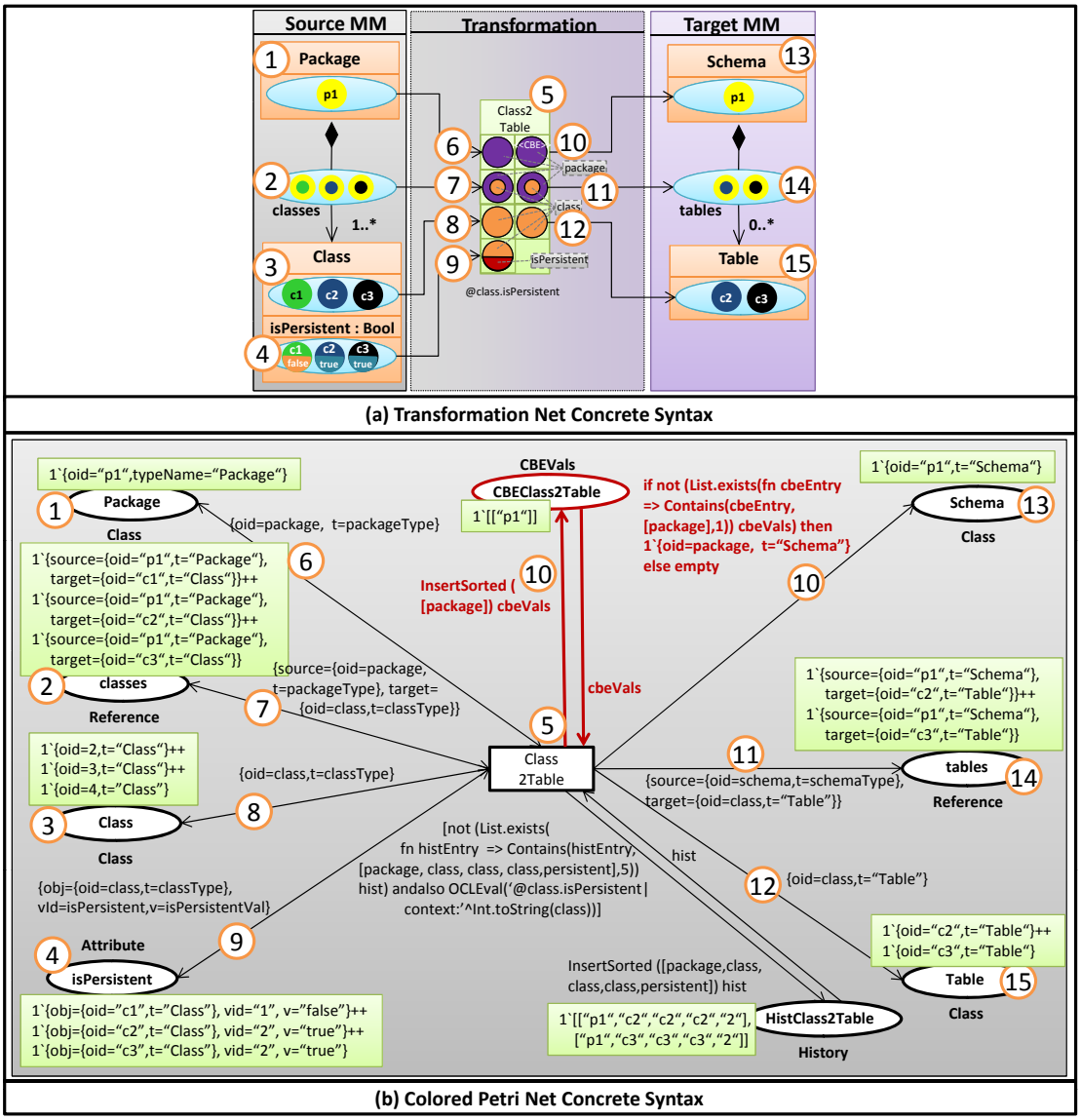


Figure 6.13: Compilation of Check Before Enforce Semantics

an empty token is produced, i.e., the current marking of the output place remains unchanged. In this respect, duplicate `Schema` instances are prohibited, i.e., only a `Table` object and an according `Package.tables` link are created. If several patterns exist which are marked as check before enforce targeting the same place, it has to be ensured that all transitions use the same `CBEVals` place in order to check if a target element was already created by a different transition. For this a so-called fusion place is used (cf. Section 6.5). Finally, if a key is specified it has to be ensured that also no values are created, which are contained in an object which is part of the key, meaning that the according arcs exhibit also the condition on the arc expression.

6.3.3 Compilation of Functions and Conditions

The example depicted in Fig. 6.7 uses a condition (cf. (17)) demanding that only persistent classes should be transformed into according tables, realized by means of an according condition `@class.isPersistent` on the Transformation Nets' transition. This condition has to be evaluated by the CPN when enabling transitions, i.e., the guard condition has to take care of the conditions specified in Transformation Nets. Nevertheless, since CPNs do not allow for OCL as inscription language, the OCL condition may not be evaluated as such in CPNs. Therefore, the evaluation of the condition is delegated to a server using the Comms/CPN library [48], by means of the function OCLEval, which is shown in Listing 6.3 (for details the reader is referred to the description of the prototype in Chapter 8). The function requires on the one hand the condition to evaluate (i.e., `@class.isPersistent`) and on the other hand the object bound by the CPN engine as actual context for the OCL expression (i.e., the value of the bound variable `class`). This information is sent to a remote server (cf. line 2 in Listing 6.3) which evaluates the expression. The result returned by the OCLEval function (cf. line 3 in Listing 6.3) is then evaluated in the guard of the CPN transition. For the evaluation of functions in Transformation Nets, the same approach is followed but instead of expecting a boolean return value, a string encoded return value is expected (since values are represented as strings only), which is accomplished by the function OCLFunctionEval, shown in the lower part of Listing 6.3. Both, OCL conditions and functions are allowed to include variables of query patterns only. This is since Transformation Nets are not allowed to read from the target model.

Listing 6.3: Functions to delegate OCL evaluation to a remote server

```
1  fun OCLEval(oclString) =
2      (ConnManagementLayer.send("OCLServer",oclString,stringEncode);
3      ConnManagementLayer.receive("OCLServer",stringDecode) = "true")
4  ------------------------------------------------------------------
5  fun OCLFunctionEval(oclString) =
6      (ConnManagementLayer.send("OCLServer",oclString,stringEncode);
7      ConnManagementLayer.receive("OCLServer",stringDecode))
```

6.4 Compilation of Inheritance in Transformation Nets

In the previous section, it was described how the dynamic parts of Transformation Nets are compiled to CPNs, but up to now, the compilation of inheriting transitions in Transformation Nets has been omitted. Since CPNs do not support the concept of inheritance, neither between

places and tokens, nor transitions, inheritance is flattened during compilation. The flattening of the inheritance hierarchy of a source model was already discussed in Subsection 6.2.2, therefore the focus is on transitions and the dynamic semantics of inheriting transitions in the following.

6.4.1 Basic Concepts, Overriding Patterns and Type Substitutability

If a transition in Transformation Nets inherits from another transition, this basically means that the behavior of a supertransition should be reused by the subtransition. The subtransition is allowed to extend this behavior or change it in a defined manner. In order to represent the semantics of inheritance, the patterns of supertransitions need to be considered when compiling a subtransition, i.e., all query and production patterns along the inheritance hierarchy have to be collected. In this respect, the inheritance hierarchy is flattened during compilation.

Overriding Patterns. In order to collect the patterns, in a first step, the tokens of supertransitions that are not *overridden* are added to the subtransitions. In a second step, the overriding patterns are collected whereby overriding of tokens is achieved by equality of variable names, i.e., the object pattern `modelElement` of the subtransition `Class2Table` overrides the object pattern `modelElement` of the supertransition `MElement2SElement` in Fig. 6.14. The

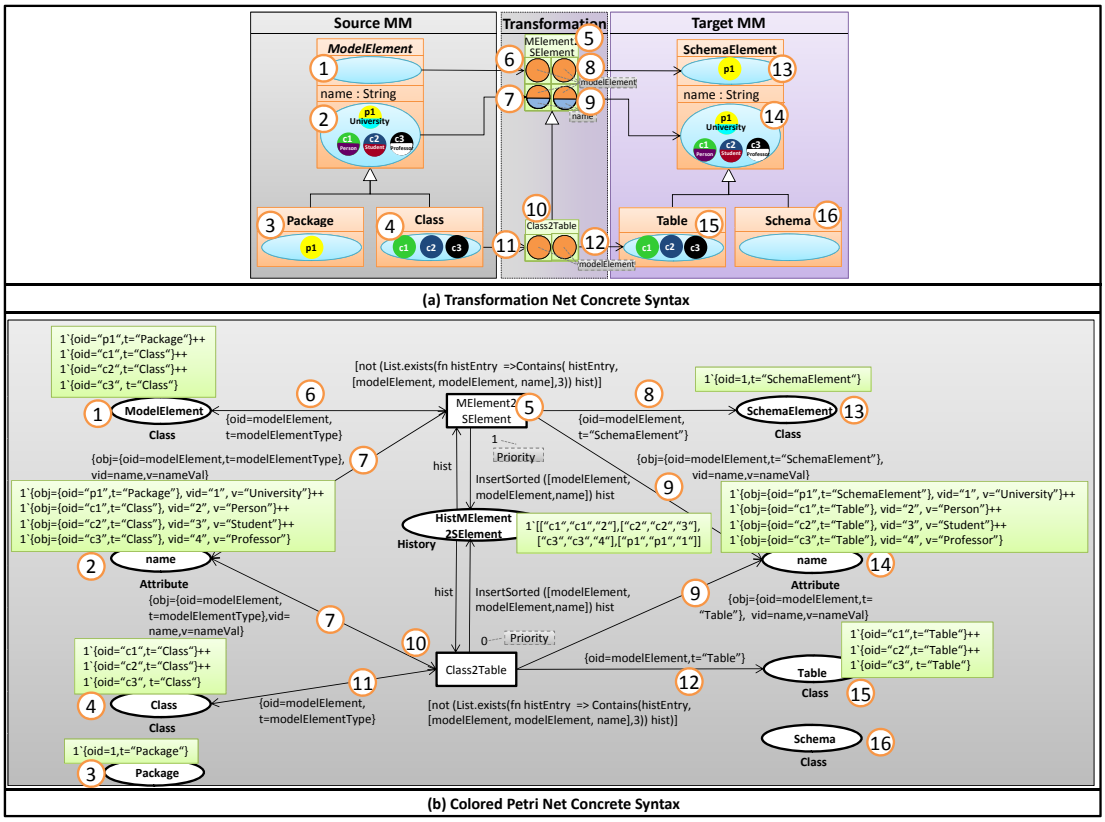


Figure 6.14: Compilation of Rule Inheritance

collection of overridden tokens depends on the chosen refinement mode. In case of the refinement mode `override` only the overriding patterns of the subtransition are taken into consideration. If the refinement mode `extend` is chosen, it is not allowed to override patterns but subtransitions might only extend the specified behavior. Consequently, an error is raised if an overridden pattern is found. Finally, in case of `inherit`, assignments of overwritten patterns are copied to the subpatterns, i.e., functions on the arc inscriptions are copied to the according arc inscription of the subtransition.

Type Substitutability. After having collected the patterns, it has to be ensured that supertransitions only transform those instances that are not affected by a subtransition. To achieve this behavior, all transitions along the inheritance hierarchy share a common history place as as can be seen in Fig. 6.14. Thereby it is important, that subtransitions and supertransitions insert the values in the same order, i.e., in our example at index 1 there is the `oid` of the model element, followed by the `oid` and the `vid` of the attribute `name`. If a subtransition adds additional patterns, the list has to be enlarged. In order to be able to compare history entries in such situations, the supertransition also enlarges its history list by adding an empty string (an example thereof is shown in Fig. 6.17 in the following subsection). Furthermore, it has to be ensured that subtransitions fire before supertransitions, since otherwise a supertransition might transform elements that could be matched by a subtransition. For this, CPNs allow to define priorities on transitions whereby a lower value means a higher priority. In this respect, it has to be ensured that a lower priority is assigned to a subtransition than to a supertransition. Consequently, the transition `Class2Table` offers a lower priority than the transition `MElement2SElement` and therefore fires first. Only if the transition `Class2Table` is not enabled any more, then the transition `MElement2SElement` is allowed to fire. In the example, first the classes `c1`, `c2`, and `c3` as well as their according names are transformed into according tables by means of the transition `Class2Table` (cf. Fig. 6.14). Afterwards the transition `MElement2SElement` is enabled for the package `p1` only (the history entries and the guard prohibits firing for the tokens `c1`, `c2`, and `c3`), which produces the according `SchemaElement p1`. By this, type substitutability is supported, being the default behavior in Transformation Nets.

Nevertheless, as described in Subsection 5.4.2, this behavior may be changed by setting the `Transition.includeSubtypes` flag to false, as can be seen in Fig. 6.15 for transition `MELement2SElement`, as indicated by the stereotype *«exclude»*. Exclusion of subtypes means that supertransitions should follow an `oclIsTypeOf` semantics, e.g., in the example only tokens of type `ModelElement` should be matched, instead of an `oclIsKindOf` semantics, e.g., tokens typed to a direct or indirect subtype of `ModelElement` should not be matched. In order to achieve this behavior in CPNs, an additional expression is added to the guard condition of the according transition, which enables a transition only if a token is of a specific type. In the example depicted in Fig. 6.15(a), the arc ⑥ originates from the place `ModelElement` and therefore, the guard in CPN specifies that the transition should only be enabled if the value of the bound variable `t` is equal to `ModelElement` (cf. `typeName="ModelElement"` in Fig. 6.15(b)). In case that several object patterns are contained by a Transformation Net transition, several expressions would be generated, which are concatenated by a logical and, i.e., every bound object has to be of a specific type only. Therefore, in case of the example depicted in Fig. 6.15 the generated target model does not contain the package instance

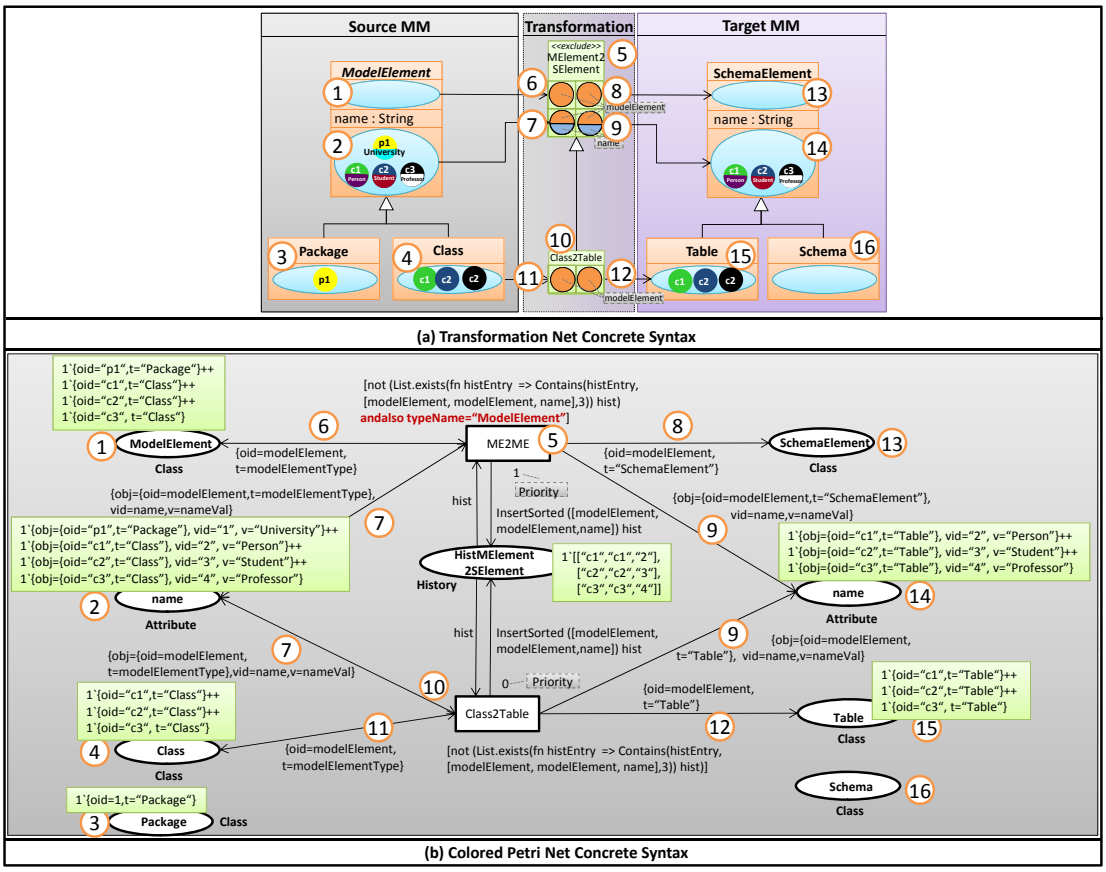


Figure 6.15: Compilation of Inheriting Transitions Excluding Subtypes

`p1` as the previous example did, since this indirect instance is not considered by the transition `MElement2SElement` any more.

Abstract Rules. As stated in Subsection 5.2.2 Transformation Nets allow to specify abstract rules in order to express that a certain rule is not executable per se, but provides core behavior that may be reused in subrules. An example thereof is depicted in Fig. 6.16(a), whereby the transition `MElement2SElement` is abstract as indicated by the stereotype *«abstract»*. Since abstract transitions are not allowed to fire and since inheritance between transitions is flattened during the compilation to CPNs anyway, abstract transitions need not to be compiled to CPNs. Therefore, Fig. 6.16(b) does not exhibit a transition for the abstract Transformation Net transition `MElement2SElement`. The target model exhibits only the classes `c1`, `c2`, and `c3` which are accordingly named (since the attribute assignment of the abstract Transformation Net transition has been inherited, cf. ⑦ and ⑨ in Fig. 6.16). Please note that the target models depicted in Fig. 6.15 and Fig. 6.16 are equal since the transition `MElement2SElement` excluding subtypes in Fig. 6.15 could never fire as no token typed to model element is available (since the class `ModelElement` is abstract). Nevertheless, in general, different target models result.

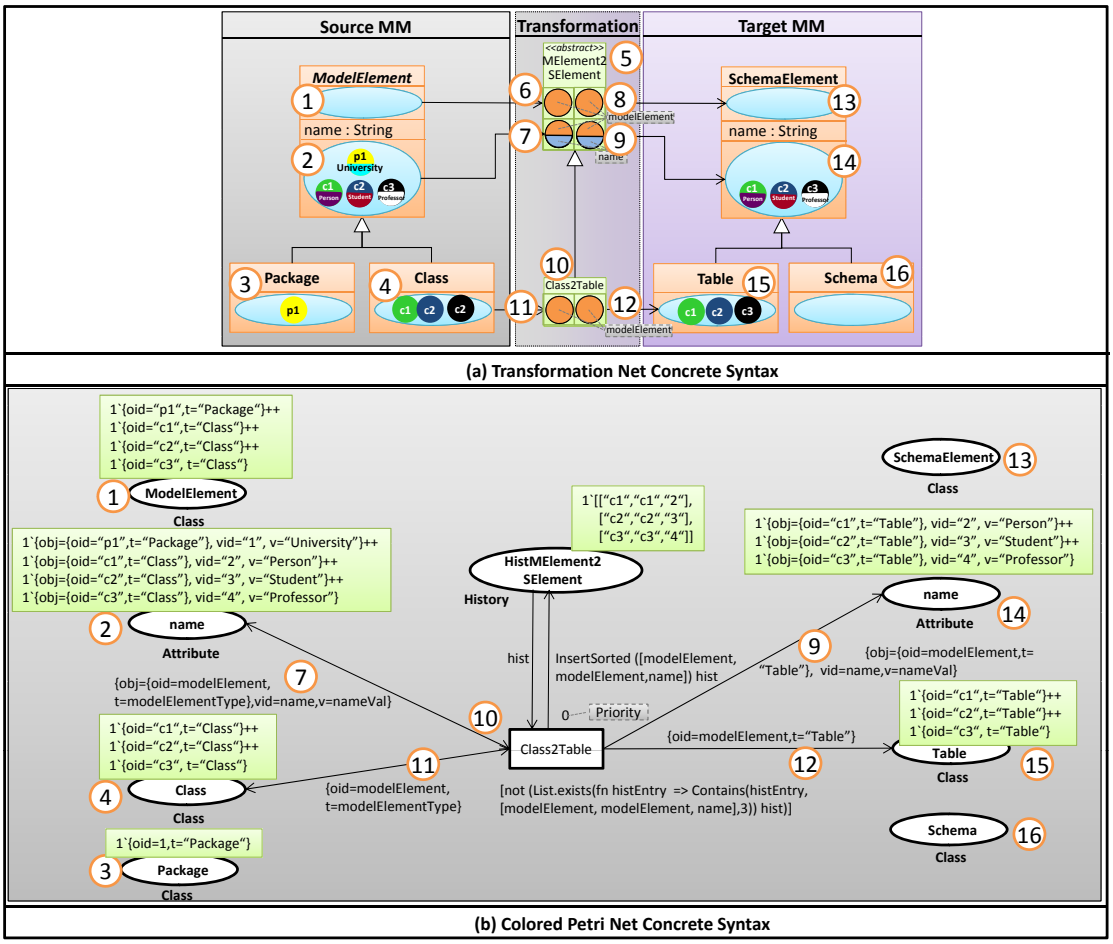


Figure 6.16: Compilation of Abstract Rules

6.4.2 Conditions and Rule Applicability Semantics

In the previous subsection, the basic concepts were presented how inheriting transitions in Transformation Nets may be compiled into according concepts in CPNs, but conditions on inheriting transitions were not considered. As stated in Subsection 5.4.2, Transformation Nets exhibit a composing behavior concerning the evaluation of conditions, i.e., all conditions along the inheritance hierarchy have to be fulfilled in order to enable the transition. Thus, conditions defined on supertransitions should also be included in the according guard of the subtransitions in CPNs. An example thereof is depicted in Fig. 6.17, whereby the guard of the transition `Class2Table` includes an expression for the condition of the supertransition (`OCLEval('@modelElement.startsWith('c') |context:'^Int.toString(modelElement))`), as well as an expression for the condition of the subtransition (`OCLEval('@cmodelElement.isPersistent |context:'^Int.toString(modelElement))`). Please note that in this example the subtransition `Class2Table` makes use of an additional query pattern for query-

ing the `isPersistent` attribute. Therefore, the history list of the subtransition would be longer, e.g., five elements in the example, than the history list of the supertranstion, which contains three elements only. Nevertheless, in order to be able to compare the lists, the history list of the supertransition is enlarged to five elements by adding two empty string values.

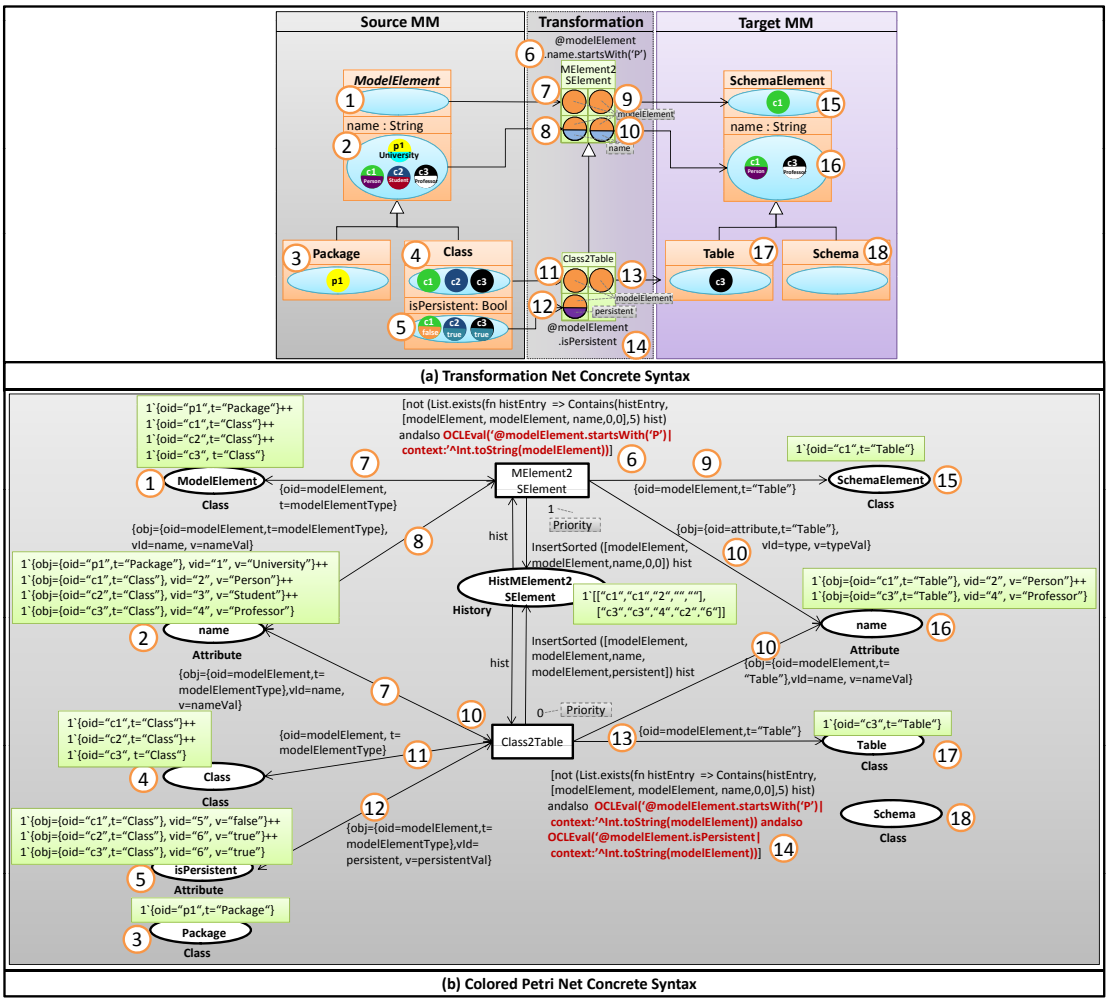


Figure 6.17: Compilation of Conditions in Inheriting Rules

When investigating the generated target model, one encounters that only the object `c3` fulfills both conditions, i.e., it is persistent and its name starts with the letter 'P' and may thus be transformed by the transition `Class2Table`. The object `c1` may not be transformed by this transition since it is not persistent, whereas object `c2` may not be transformed as it does not fulfill the condition on the supertransition `MElement2SElement`, i.e., its name does not start with the letter 'P'. On investigating the supertransition, it may be seen that it is not able to transform object `c2` either, since the condition fails. Nevertheless, it is able to transform object `c1` since it fulfills the conditions of the supertransition and has not been transformed by the subtransition

already. The transition `MElement2SElement` transforms the object `c1`, resulting in a target object typed to `ModelElement`. In this respect, Transformation Nets support rule applicability semantics (as described in Subsection 5.4.2) concerning the evaluation of conditions.

6.5 Compilation of Modules

In the previous sections the compilation of the core concepts as well as of inheritance between transitions have been presented. What has been omitted up to now is the compilation of modules, which is the focus of the following subsections. First of all, modularization concepts of CPNs, i.e, Hierarchical Colored Petri Nets, are presented. Then, second the modularization concepts of Transformation Nets are formalized to lay the basis for third, the compilation of modules to hierarchical CPNs.

6.5.1 Hierarchical Colored Petri Nets

In order to modularize CPNs, a CPN may be divided into several *modules* offering defined interfaces, which play a similar role as modules in programming languages. CPNs that make use of modules are called *hierarchical CPNs*. As can be seen in Fig. 6.18, a CPN consists of at least one module being the root of the net. A module might contain an arbitrary number of other modules, e.g., in the example the root module contains the modules `Producer` and `Consumer`. Such sub-modules are referred by the parent module by means of a so-called substitution transition (cf. `P` and `C` in the example). The interfaces of submodules are defined by means of *sockets* and *ports*. Ports exhibit so-called *prototype tags* which define whether the port is an *In* port, an *Out* port or an *In/Out* port. A socket, i.e., a place on the parent module, is assigned to a port, i.e., a place at the submodule, which is only allowed if the places exhibit an equal color-set. The socket place `p1` is assigned to the port place `source` and the socket

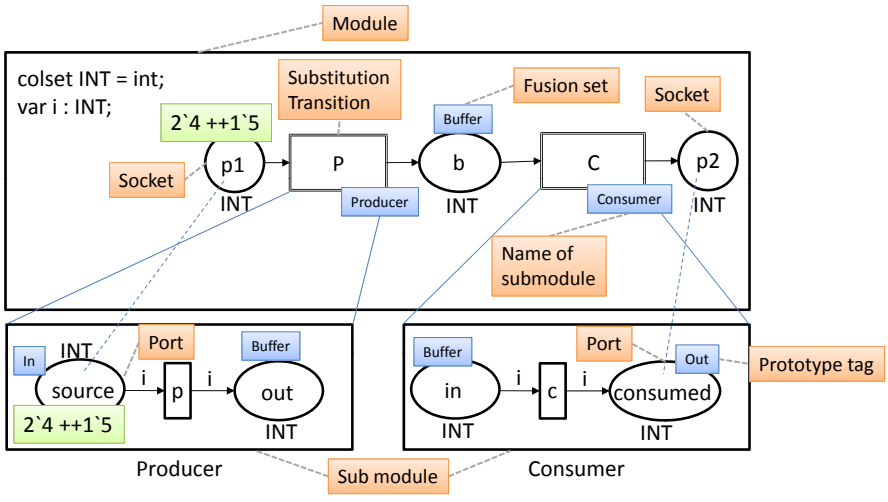


Figure 6.18: Sample Hierarchical CPN

place `p2` to the port place `consumed`. Thus, sockets and ports are used to stream tokens between places on different modules, whereby a socket and its according port always exhibit the same marking. Another possibility to exchange tokens between different modules are so-called `fusion sets`. Fusion sets allow places in different modules to be glued to one compound place across the hierarchical structure of the model and are similar to global variables known from programming languages [72]. In the example in Fig. 6.18, a fusion set `Buffer` is defined to exchange data between the `Consumer` and `Producer` module, i.e., the place `out`, `in` and `b` are members of the fusion set.

To be more formal, a hierarchical CPN is defined as a 4-tuple $CPN_H = (S, SM, PS, FS)$ in [72, p. 135]. S is a finite set of modules, whereby each module consists of a non-hierarchical CPN, extended with a set of substitution transitions being a subset of all transitions, i.e., $T_{sub} \subseteq T$, a set of port places, i.e., $P_{port} \subseteq P$, and a port type function $PT\colon P_{port} \rightarrow \{IN, OUT, I/O\}$ that assigns a prototype tag to each port place. $SM\colon T_{sub} \rightarrow S$ assigns a submodule to each substitution transition. PS is a port-socket relation function that assigns a port to a socket and FS is a set of non-empty fusion sets (cf. [72, p. 136] for details).

6.5.2 Formalization of Modules in Transformation Nets

In order to define the relationship between modules in Transformation Nets and modules in hierarchical CPNs, modules of Transformation Nets are formally defined in the following. According to the definition of hierarchical CPNs, a Transformation Net with modules is a 5-tuple $TransformationNet_{mod} = \{M, PO, \Sigma_{Port}, PT, PA\}$. M defines a non empty set of modules, which contain Transformation Nets. PO is a finite set of ports and Σ_{Port} is a set of predefined port types, i.e., $\Sigma_{Port} = \{Class, Attribute, Reference, TracePlace\}$. $PT\colon P \rightarrow \Sigma_{Port}$ is a color-set function that assigns a color-set to each port, i.e., each port is again typed to either `Class`, `Attribute`, `Reference` or `TracePlace`. $PA \subseteq P \times PO \cup PO \times P \cup PO \times PO$ is a set of directed arcs that connects either ports and places or two ports such that Type[P]=Type[PO] or Type[PO]=Type[PO]. In this respect, sockets and ports of hierarchical CPNs are both represented by means of ports in Transformation Nets. Furthermore, ports in Transformation Nets are explicitly connected via arcs. The details of the compilation are shown in the following subsection.

6.5.3 Compilation of Modules to Hierarchical CPNs

In order to represent the modularization concepts of Transformation Nets in CPNs accordingly, Transformation Nets using modules are compiled into hierarchical CPNs following the above formal definitions.

Blackbox View. The example depicted in Fig. 6.19 makes use of two modules (cf. ⑥ and ⑩) in order to translate `Packages` to `Schemas` and `Classes` to `Tables`. The interfaces of the modules are described by means of ports (cf. ⑦ to ⑨ and ⑪ to ⑬ in Fig. 6.19(a) and (b)), whereby places and ports are connected by means of arcs. The resulting hierarchical CPN is depicted in Fig. 6.19(c) and (d). Every instance of a Transformation Net `Module` is translated to an according instance of a substitution transition (cf. ⑥ and ⑩ in Fig. 6.19(c), whereby `Instances` represent substitution transitions in the abstract syntax – as may be seen in ⑥ and

⑩ in Fig. 6.19(d)). Furthermore, substitution transitions refer to submodules by means of the attribute `Instance.subpageID`.

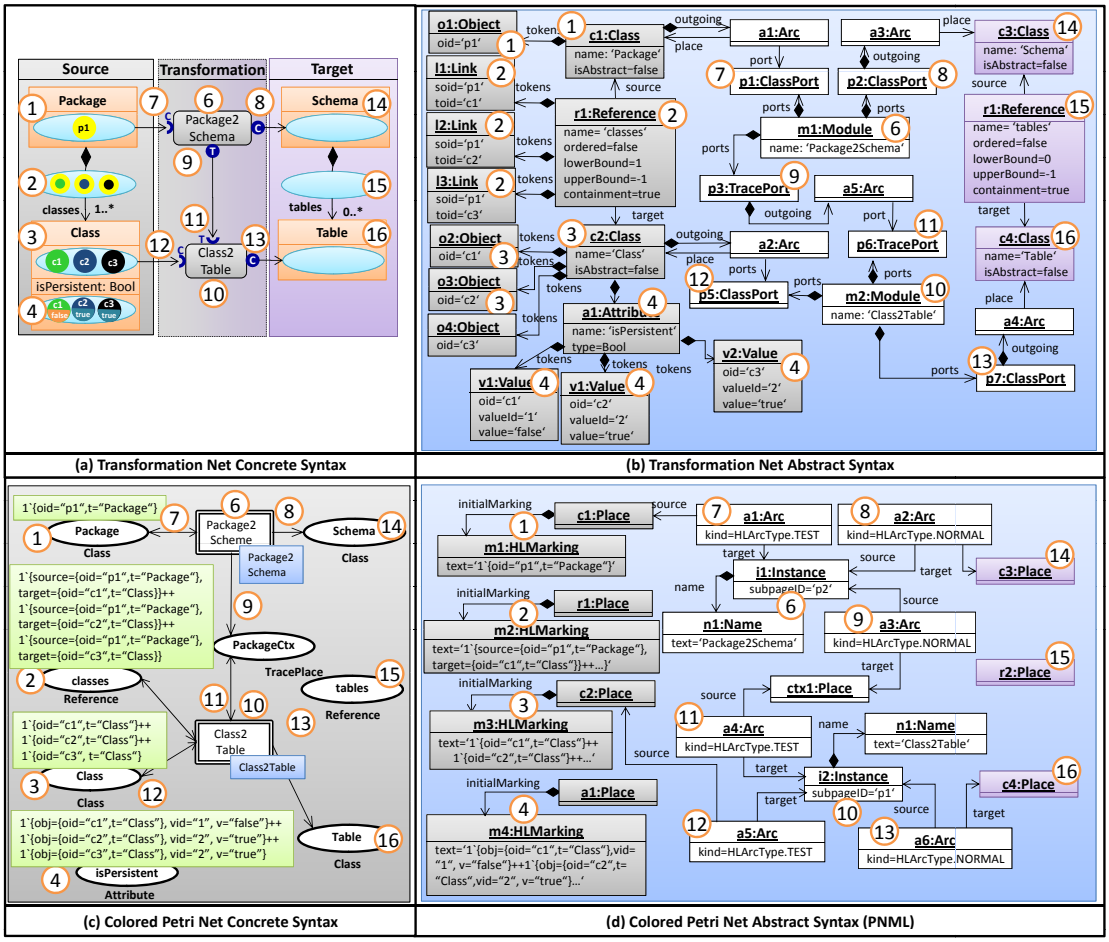


Figure 6.19: Compilation of Blackbox View

Whitebox View. The actual transformation logic is hidden in a module's whitebox view in Transformation Nets or in submodules in CPNs. In this respect, Fig. 6.20 depicts the whitebox view of the module `Package2Schema` and the corresponding submodule in CPNs, both in concrete and abstract syntax. Transformation Nets narrow the scope to the involved metamodel elements first, i.e., only the metamodel classes `Package` and `Schema` are shown in the whitebox view since they are required by the according transition. In order to represent the whitebox view in CPNs an according submodule has to be created. Thereby, submodules are represented by means of `Page` instances in the abstract syntax. The `Net n1` contains two pages[4], whereby `page1` contains the elements of the blackbox view, i.e., places representing the metamodel elements as well as the substitution transition (cf. ② and ⑥ in Fig. 6.20(d)). `Page2` contains the

[4]The page for the substitution transition `Class2Table` has been omitted due to reasons of brevity.

elements of the module for the substitution transition `Package2Schema`. The ports of a Transformation Net module are compiled to ports of submodules in CPNs, which are represented by means of so-called `RefPlaces` in the abstract syntax (cf. ③ - ⑤ in Fig. 6.20(d)). In order to define the required socket/port assignments, i.e., which place on the parent page corresponds to which place on the subpage, so-called `ParameterAssignments` have to be defined. Thereby, the value of the attribute `ParameterAssignment.parameter` refers to the id of the socket place (place on the parent page), and the value of the attribute `ParameterAssignment.value` to the id of the port (place on subpage). Please note that the history place `HistPackage2Schema` has been changed from a "normal" place to a place being a member of a fusion set. This is since it is possible in Transformation Nets to inherit between transitions which are encapsulated in different modules and therefore history places in CPNs need to be shared by transitions on different pages. Furthermore, in Transformation Nets the history concept is hidden from the user. The design rationale was not to introduce ports for history places but to use fusion sets instead, in order to hide the history concept from the user in CPNs at first sight as well and to gain the same structure in CPNs as in Transformation Nets, i.e., equal number and types of ports. In the abstract syntax, fusion sets are represented by instances of the class `FusionGroup`, whereby the reference `FusionGroup.references` refers to the according members of the fusion group (cf. ⑦ in Fig. 6.20).

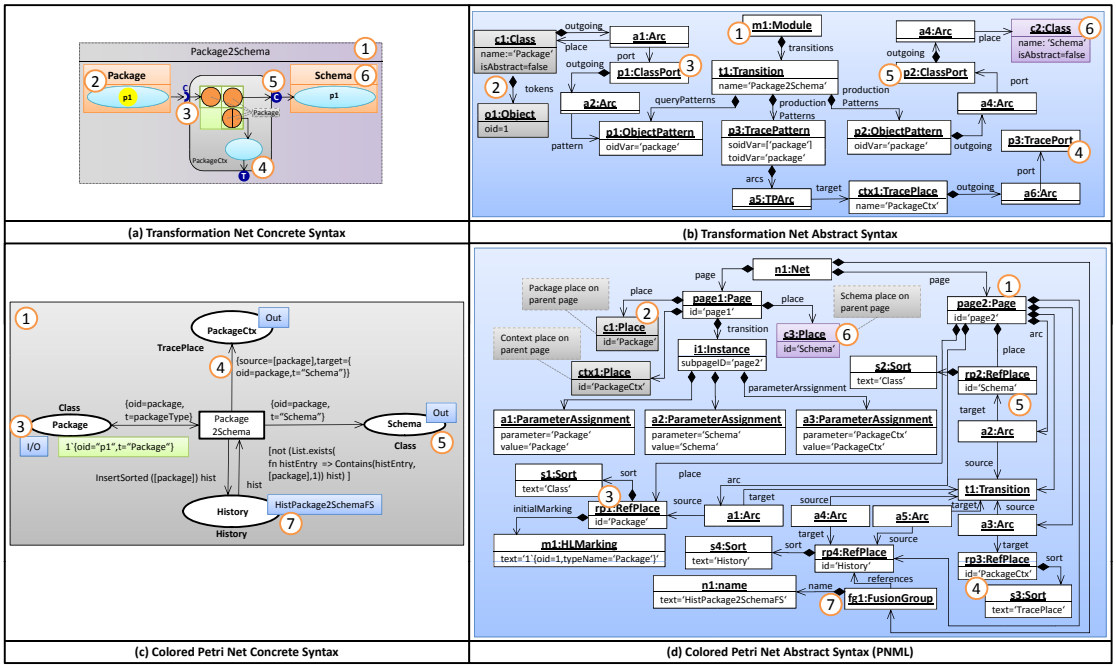


Figure 6.20: Compilation of Whitebox View

6.6 Summary

In summary, this chapter presented the compilation of Transformation Nets to CPNs. In a first step, the concepts of CPNs were introduced by example and by means of the formal definitions thereof. In order to align the formal concepts of CPNs with Transformation Nets, the concepts of Transformation Nets have been formalized as well, building the basis for the actual compilation which was presented in detail. Thereby, the translation of places and tokens was discussed before the focus was shifted to the transformation logic. First, it was mainly shown how patterns may be represented by according arc inscriptions. Second, it was discussed how to realize the non-consuming behavior of Transformation Nets in CPNs by introducing so-called history places. Third, it was shown how OCL conditions and functions are compiled into according concepts in CPNs and it was discussed how the evaluation thereof may be done, although CPNs do not support OCL as inscription language.

After presenting the core concepts of Transformation Nets, the compilation of reuse mechanisms, i.e., inheritance and modules, was discussed. Since CPNs do not support inheritance, the concept of inheritance in metamodels and between transitions in Transformation Nets has to be flattened during compilation. It was shown, how the requirements on the dynamic semantics of inheritance in Transformation Nets posed in Subsection 5.4.2, may be accordingly represented in CPNs. The chapter finished with a discussion on how to represent the module concept of Transformation Nets in CPNs. For this purpose, hierarchical CPNs have been introduced and the compilation of modules to hierarchical CPNs has been shown. Thus, this chapter showed how standard CPNs may be used to as a semantic domain for Transformation Nets, additionally building the basis for sophisticated debugging facilities, which is the focus of the following chapter.

Chapter 7

Debugging Support for Model Transformations

I have not failed.
I've just found 10,000 ways that won't work.

— *Thomas A. Edison*

Contents

In the previous chapters Transformation Nets have been introduced as a runtime model for model-to-model transformations and their translation to CPNs, as a semantic domain was discussed. The introduction of a dedicated runtime model allows for sophisticated means of debugging, which is the focus of this chapter. Thereby, dedicated support for every of the three debugging phases proposed in [173], being (i) observing facts, (ii) tracking origins, and (iii) fixing failures, is provided, as can be seen in Fig. 7.1. In addition to PaMoMo contracts, Transformation Nets allow for further means to observe facts in a transformation specification. First, since Transformation Nets provide an integrated view on model transformations, i.e., the meta-models, models and transformation logic are explicitly represented, code-smells may be detected by the transformation designer, i.e., structures in Transformation Nets that are likely to lead to a failure. Second, formal properties of CPNs, e.g., termination or confluence, may be used to

observe errors. The phase of tracking origins is supported by (i) simulation-based debugging, i.e., executing the transformation specification in a stepwise manner, (ii) query-based debugging, i.e., employing OCL queries on the runtime model, also enabling forensic debugging, i.e., analyzing an already executed transformation to try to reason for failures, and (iii) property based debugging, i.e., using CPN properties (in addition to common properties like termination) for debugging. Finally, means for fixing a failure in Transformation Nets are discussed, i.e., changing the model and the transformation logic.

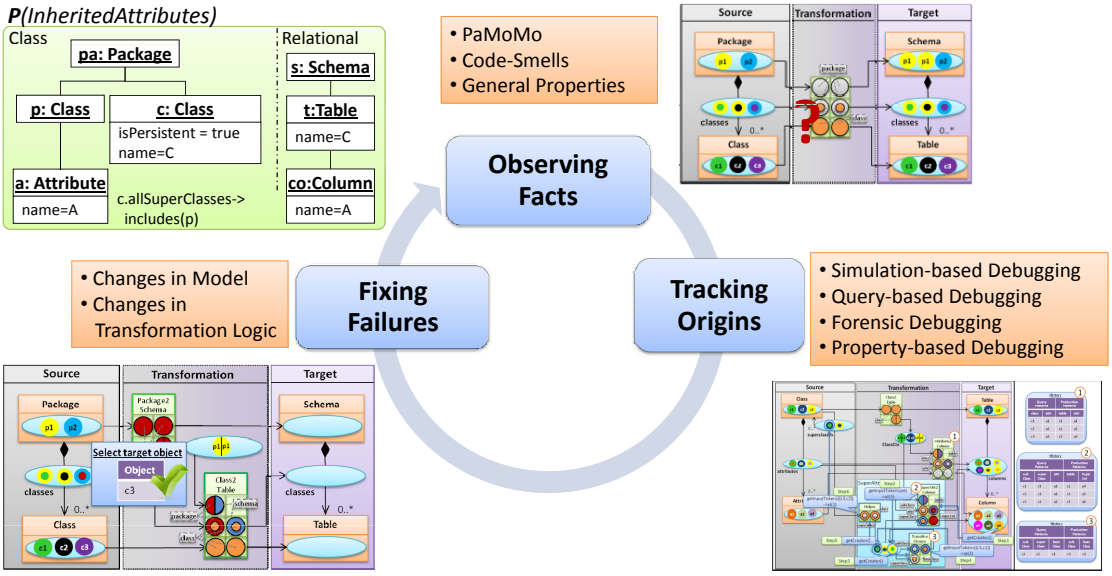


Figure 7.1: Overview on Debugging Phases and Support in Transformation Nets

7.1 Code-Smells in Model Transformations

As stated above, Transformation Nets may act as a common runtime for declarative, model-to-model transformation languages in order to provide dedicated means for testing and debugging. The integrated view on model transformations, i.e., the representation of metamodels and models together with the actual transformation logic, allows to detect potential failures. For example, if a transformation specification is translated to Transformation Nets, the static structure might already indicate failures. In the following, common code-smells, i.e., structures that are likely to cause failures, are discussed and it is explained how they may be detected by inspecting the structure of Transformation Nets. Fig. 7.2 classifies potential code-smells into intra-transition code-smells, i.e., code-smells that concern a single transition only, and inter-transition code-smells, i.e., code-smells concerning the interplay between transitions. In the following, each category is described in a pattern like style, whereby foremost the problem is described succeeded by a hint, how to spot these code-smells in Transformation Nets.

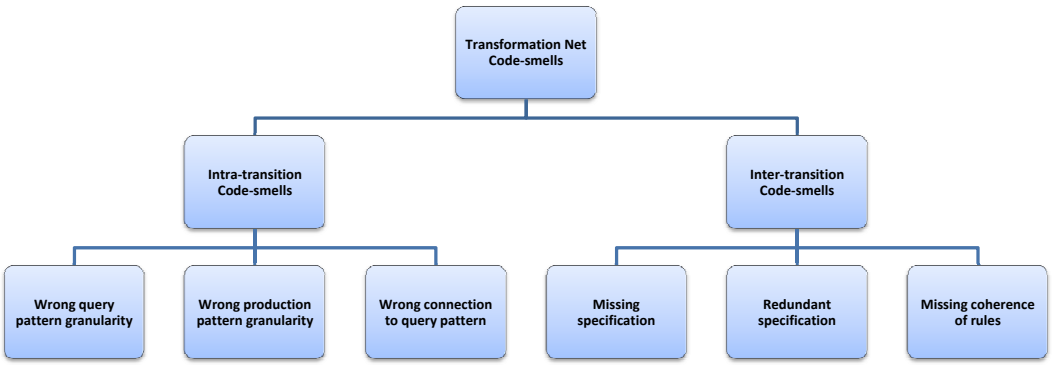


Figure 7.2: Taxonomy of Common Code-Smells in Transformation Nets

7.1.1 Intra-Transition Code-Smells

Pitfalls within a single transition might concern either the matching phase of input elements, or the the production phase of the desired target elements.

- **Wrong Query Pattern Granularity.** A common code-smell concerns the pattern granularity, i.e., the number of 1:n relationships occurring in the precondition (LHS) side of a transition. Starting from an object, only one 1:n relationship is preferred (e.g., between `Package` and `Class`), since a further 1:n relationship (e.g. between `Class` and `Attribute`) could either lead to too many matches or to too few matches which is not intended, as can be seen in Fig. 7.3(a). In this example, on the one hand the package `p1` is matched twice because the contained class `c1` contains two attributes `a1` and `a2`, since no check before enforce semantics was applied, and therefore, too many packages are created. On the other hand, no schema is produced for the package `p2`, since the class `c2` does not contain any attributes, which is probably not intended as well.

  Detection in Transformation Nets: A wrong pattern granularity may be statically checked by inspecting the involved query reference pattern. If a transition contains more than one reference pattern and if the according source reference places offer an upper bound greater than one, then a warning is given to the transformation designer. This is ensured by an according OCL invariant on the transition (cf. Listing 7.1). If the transformation specification is already executed, a wrong granularity may also be detected if the target places or the trace places of a certain transition contain duplicates, i.e., tokens with the same color, which again is a strong indication that there were too many matches.

Listing 7.1: Invariant to Check Granularity of Query Reference Patterns

```
1  context Transition inv QueryPatternGranularity:
2  self.queryReferencePatterns->collect(inArc)-> flatten()
3  --check if the upper bound of the reference is greater than 1
4  ->collect(x : PTArc | x.source.upperBound > 1)->size() <= 1
```

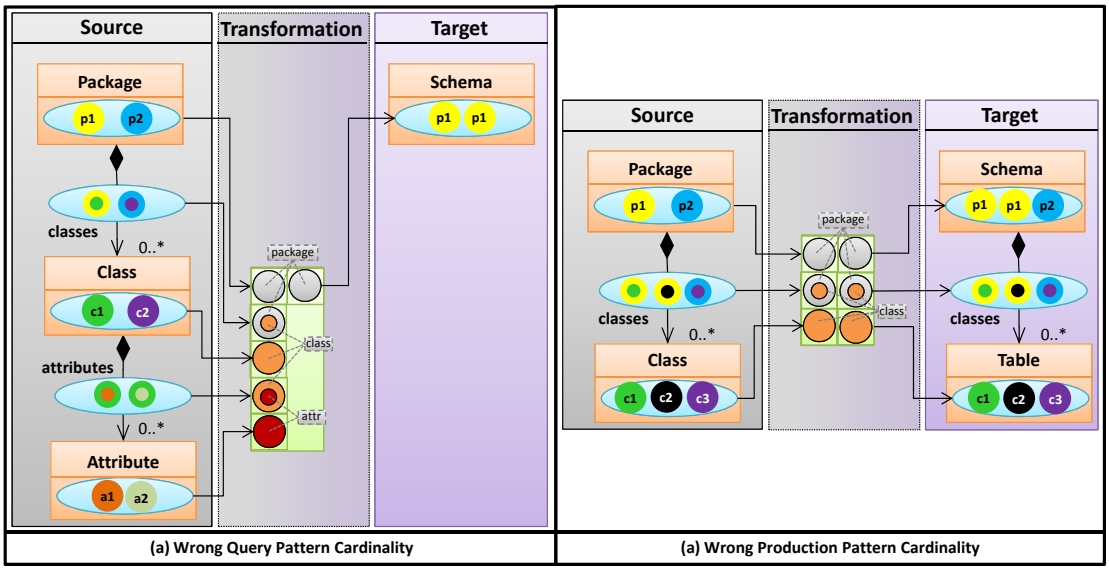


Figure 7.3: Wrong Pattern Granularity

- **Wrong Production Pattern Granularity.** At the target side, it is typically not desired to produce source and target objects as well as the link of an 1:n relationship by a single rule, since this would lead to too many source objects of the link (if the rule does not support a *check before enforce* semantics). As can be seen in Fig. 7.3(b), too many packages are created since `p1` is matched twice (i.e., in the combination `package = p1` and `class = c1` and in the combination `package = p1` and `class = c2`). Therefore matching 1:n references and producing 1:n references within a single transition should be omitted.

  Detection in Transformation Nets: On the one hand this code-smell may be detected by means of an OCL invariant if there exists a target arc to the source class place of a reference and to the target place of a reference, e.g, the places `Schema` and `Class` in the example, and if there exists a query pattern which matches for an unbounded reference, e.g., `classes` in Fig. 7.3. On the other hand, if the Transformation Net is already executed, again duplicate tokens may be found in the source place of the reference. Both facts are a strong indication that the production patterns offer the wrong granularity and therefore an according warning is raised.

Listing 7.2: Invariant to Check Granularity of Production Reference Patterns

```
1  context Transition inv ProductionPatternGranularity:
2  self.queryReferencePatterns->collect(inArc)-> flatten()
3  --check if pattern exists with upper bound greater than 1
4  ->collect(x : PTArc | x.source.upperBound > 1)->isNotEmpty() implies
5  --check if source variable is also used in an object pattern
6  self.productionReferencePatterns->forAll( x |
7  not self.productionObjectPatterns
8  ->select(y | not y.cbe)->flatten()->collect(z |
9  z.oidVar)->includes(x.soidVar))
```

- **Wrong Connection to Query Pattern.** A further code-smell may occur if the production pattern is wrongly connected to the source domain pattern by missing or incorrect variable assignments, i.e., a new instance of an object is generated instead of copying the source element.

  Detection in Transformation Nets: By comparing the colors of the query patterns on the left side with the colors of the production patterns on the right side of a transition, it may be seen if new objects, values, or links are created, i.e., these objects have not been bound by variables. Nevertheless, since this might be intended in several situations, e.g., when merging several source objects to a single target object, this situation does not raise a warning.

7.1.2 Inter-Transition Code-Smells

Inter-transition code-smells, i.e., suspicious structures concerning the interplay of the specified transitions, deal with *metamodel coverage*, i.e., if all elements of the involved metamodels are affected by transitions, *redundancies*, i.e., if metamodel elements are affected by several transitions, and finally *coherence* between transitions.

- **Missing Specification.** If no rule matches a certain metamodel element, then this element will not participate in the transformation process and the according instances will not result in any target instances, which leads to information loss during the transformation. As on the source side, also on the target side metamodel elements may not be targeted by a single rule and therefore no according instances may be created.

  Detection in Transformation Nets: If no source arc originates from a certain source place (cf. Fig. 7.4 where no arcs originate from the class `Attribute`), this metamodel element will not be considered in the transformation. The same is true on the target side, i.e., if no arc targets a certain place representing an element of the target metamodel, instances of this metamodel element may not be created by the transformation. Both code-smells are automatically detected by means of OCL invariants (cf. Listing 7.3) in Transformation Nets leading to according warnings.

Listing 7.3: Invariant to Check Metamodel Coverage

```
1  context LHS inv SourceMMCoverage:
2  self.places->forAll(p | p.outgoing->notIsEmpty())
3  ------------------------------------------------
4  context RHS inv TargetMMCoverage:
5  self.places->forAll(p | p.incoming->notIsEmpty())
```

- **Redundant Specification.** If more than one rule matches a certain metamodel element, then this may lead to redundant elements on the target side, unless according conditions match for disjoint subsets. Again, the same applies to the target metamodel, i.e., if elements of the target metamodel are targeted by more than one rule, then it may happen that these parts will be produced several times (if no check before enforce semantics is employed).

Detection in Transformation Nets: On the source side, this code-smell may be detected if several arcs originate from one source place. On the target side, this may be detected if more than one arc targets a certain place. Again this may be automatically checked by employing OCL constraints, leading to according warnings in Transformation Nets (cf. Listing 7.4). Nevertheless, since redundant specifications are common practice in some transformation languages, e.g., QVT Relations, the validation of this code smell can be turned off in the prototype (cf. Section 8.2). Furthermore, if the Transformation Net is already executed and if a target place contains duplicates, i.e., same-colored tokens, this indicates redundantly specified parts as well. As the example in Fig. 7.4 depicts, the module `PackageClasses2SchemaTables` and the module `Class2Table` target the class place `Table`, which leads to a redundant generation of table instances.

Listing 7.4: Invariant to Check for Redundant Specifications

```
1  context LHS inv SourceMMRedundancy:
2  self.places->forAll(p | p.outgoing->size() > 1)
3  ------------------------------------------------
4  context RHS inv TargetMMRedundancy:
5  self.places->forAll(p | p.incoming->size() > 1)
```

- **Coherence Between Rules.** Typically, rules in transformation languages interact with each other, i.e., the result produced by one rule allows other rules to transform their corresponding elements (cf. trace and intermediate places). In this respect, rules are typically using trace information or explicit calls, i.e., when and where clauses in QVT Relations to synchronize each other. If several unrelated rules are specified, then they work entirely independently of each other, resulting in unconnected parts in the output model. This may be intended, in case that the input model consists of unconnected parts too, but normally this is not intended, especially in the context of Ecore, which demands a tree structure of the model elements and at least a connection to the rule transforming the root container, has to be established. Therefore, also the according Transformation Net requires traces places to allow for interaction between transitions.

  Detection in Transformation Nets: In Transformation Nets, transitions interoperate via trace places and modules are connected via according trace ports. Therefore, if transitions or modules are not connected via according trace places, the transitions are independent of each other. Thus, the `Table` instances produced by the module `Class2Table` are not linked to their according packages, as may be seen by the number of tokens in the reference place `tables` in the example in Fig. 7.4. In order to provide hints to the transformation designer, the target metamodel can be analyzed. In this respect, if a transition targets a certain class and if this class offers a containment reference to another class, then the transitions that target this class have to interoperate.

- **Invalid Target Model.** The generated target model of a transformation must again conform to its according metamodel. For example, dangling references must not occur, i.e., links have to point to a valid target object. An incorrect target model may result from the fact that in Transformation Nets it is possible to match for arbitrary elements, e.g., to match for a value, and to produce an arbitrary element thereof, e.g., a link. This is

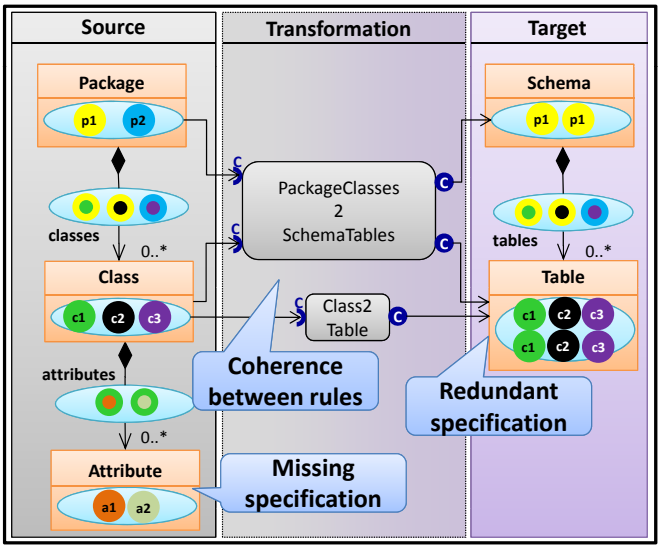


Figure 7.4: Inter-Transition Code-Smells

especially favorable to overcome structural heterogeneities [94], but may induce incorrect target models. Nevertheless, this may also happen in other Transformation Languages, e.g., in graph transformations, objects may be deleted which are still referred by some links.

Detection in Transformation Nets: Invalid configurations of a target model may be detected by inspecting the tokens of the generated target model, i.e., the colors of links and the object color of values has to be present in the according class places. In order to ensure such well-formed constraints, OCL invariants are added to the target places. For example, for all tokens in attribute places it is checked if there exists an according object token. If this is not the case, these tokens are highlighted, which may serve as starting point for debugging. A similar constraint is put onto reference places which checks if the according source and target objects exist. Additionally, boundedness constraints are validated, i.e., if the number of tokens that originate from a certain link token (outer color) does not exceed the specified upper-bound of the reference.

Listing 7.5: Invariants to Check if Generated Target Model is Correct

```
1  context Attribute inv HasObject:
2  self.class->tokens->collect(oid)->
3  includesAll(self.tokens->collect(oid))
4  ---------------------------------------------------
5  context Reference inv HasSource:
6  self.source->tokens->collect(oid)->
7  includesAll(self.tokens->collect(soid))
8  ---------------------------------------------------
9  context Reference inv HasTarget:
10 self.target->tokens->collect(oid)->
11 includesAll(self.tokens->collect(toid))
12 ---------------------------------------------------
```

```
13          context Reference inv CheckUpperBound:
14          self.upperBound <> -1 implies self.tokens->forAll(x: Token |
15          self.tokens->collect(x.oid)->size() <= self.upperBound)
16          ------------------------------------------------------------
17          context Reference inv CheckLowerBound:
18          self.lowerBound <> 0 implies self.tokens->forAll(x: Token |
19          self.tokens->collect(x.oid)->size() >= self.lowerBound)
```

In summary, the static structure of Transformation Nets may already indicate certain failures in the transformation specification, providing a potential starting point for debugging. In the following, means for debugging in Transformation Nets are discussed in detail.

7.2 Simulation-Based Debugging

Although the static analysis of Transformation Nets and the corresponding detection of code-smells may point to failures, often failures in transformation specifications may only be detected by means of live-debugging, i.e., simulation of the execution. The simulation of the transformation specification allows a transformation designer to get an insight into the specification, i.e., the hidden operational semantics is made explicit, in order to foster debugging. In this respect, Transformation Nets provide various means to support the transformation designer in live-debugging in order to effectively find the origin of a failure. Following the classification presented in Section 2.3, the according means in Transformation Nets are discussed. Thereby, (i) means for selecting a certain part of the transformation code, (ii) means for inspecting the current execution state and, finally, (iii) means for investigating the dynamic behavior are presented.

7.2.1 Selection

Transformation Nets allow (i) to select an enabled transition and according bindings, i.e., debugging of the matching process, and (ii) to set breakpoints on different elements (i.e., transitions, places, tokens), in order to provide flexible means to the transformation designer to select a certain starting point for debugging.

7.2.1.1 Debugging of the Matching Process

As stated in Section 2.3, debugging support during the matching phase of a model transformation is of utmost importance. Since transformation engines typically select applicable rules non-deterministically, the debugging environment needs to accordingly visualize the rules that are currently applicable. Since Transformation Nets build a DSL on top of CPNs, which inherently support non-determinism, the enabled transitions have to be made explicit. For this, the according enabled transitions are highlighted (cf. transitions with a bold, green border in Fig. 7.5), indicating that (only) these transitions may be fired. In the example in Fig. 7.5 both transitions are enabled, since there already is a trace token available which enables the transition `Class2Table`. Consequently, the transformation designer is allowed to chose an arbitrary transition to fire. If an enabled transition is contained within a module, the blackbox view of the module is accordingly highlighted as well.

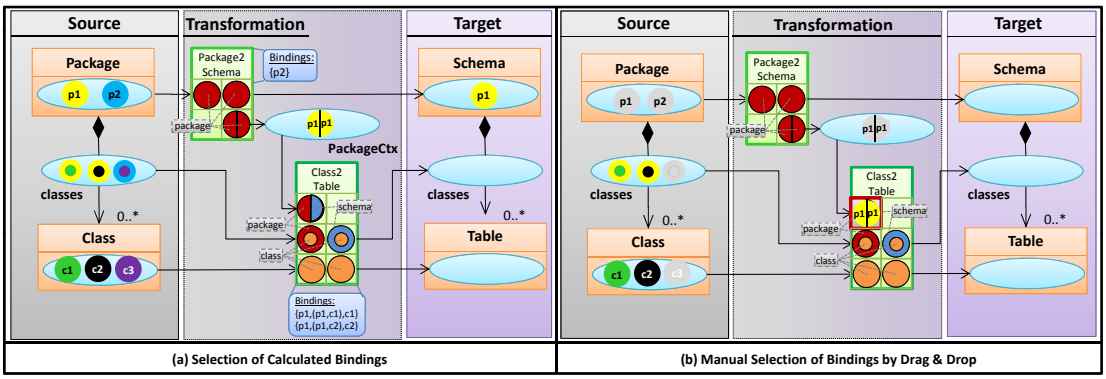


Figure 7.5: Debugging Support in the Matching Phase

If a transition is enabled, it may be the case that there exist several valid bindings. Thus, the transformation designer should be enabled to select a desired one. Transformation Nets support this scenario by two different mechanisms being (i) selection of calculated bindings and (ii) user-defined bindings. Concerning the first mechanism, every enabled transition may be asked for its currently possible valid bindings, which are presented to the transformation designer and from which he is allowed to select one, as can be seen in Fig. 7.5(a). Concerning the second mechanism, the transformation designer may drag and drop tokens from source places to a query pattern of the according transition. The transition checks, if the specific token is part of a valid binding. If this is the case, the query pattern is bound to the according token. Additionally, the remaining tokens in source places that may not be bound any more are greyed out and may not be dropped onto query patterns of the according transition anymore (cf. Fig. 7.5(b)). On the one hand, the transformation designer may then complete the binding by dragging and dropping the remaining tokens to the according query patterns. On the other hand, the matching may also be auto-completed, i.e., the transformation designer might select from the remaining calculated bindings, meaning that only those bindings are presented that contain the already bound tokens. If the first selected token is not part of a valid binding, an according error message is presented, stating the reason why a certain token may not be bound, e.g., if it does not fulfill a certain condition.

7.2.1.2 Breakpoints

Whereas in common programming languages a breakpoint is typically set to the desired line of code, in Transformation Nets breakpoints may be set on four different types of metamodel elements, being (i) transitions, (ii) modules, (iii) tokens and (iv) places, as can be seen in Fig. 7.6(a). Breakpoints on transitions are closest to those known from programming languages. Per default, the execution of Transformation Nets stops at this kind of breakpoint every time the according transition is enabled. Nevertheless, the transformation designer might change this behavior and may configure the breakpoint such that it stops execution every time a certain transition is *not* enabled. The same principle is followed concerning modules, whereby execution is stopped

every time anyone of the contained transitions is enabled or disabled depending on the configuration of the breakpoint. If a breakpoint is attached to a certain token, execution is stopped if this token is successfully bound to a transition, i.e., if it is part of a valid binding. Finally, concerning places, execution is stopped either if a token is about to be read from a certain source place or if a token is going to be put into a certain target place.

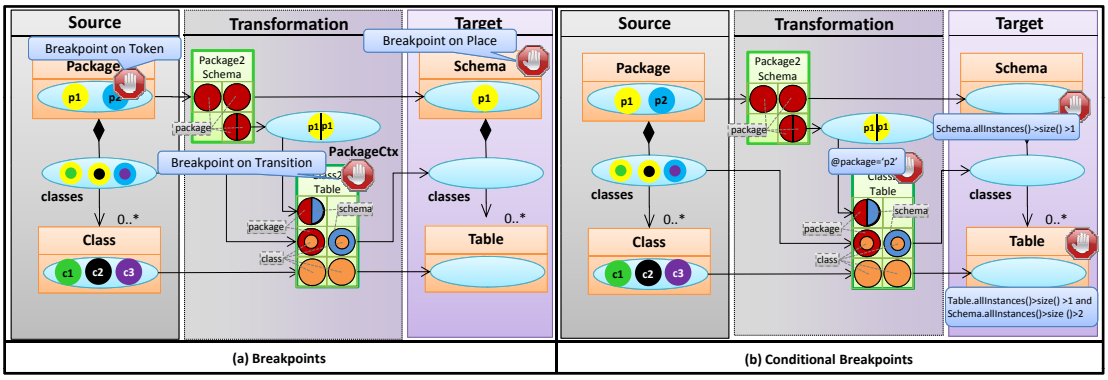


Figure 7.6: Breakpoints in Transformation Nets

Conditional Breakpoints. Although breakpoints may be attached to different elements, they are not very flexible, i.e., the execution might stop too often, since it might not exactly be the situation the transformation designer wants to debug. Therefore, conditional breakpoints are provided, i.e., OCL expressions are used to further restrict the applicability of a certain breakpoint. Conditional breakpoints may again be specified at different levels of granularity. Thus, it may not only be defined that execution should stop, e.g., if a certain token is streamed into a certain place, i.e., *local condition*, but also if a certain combination of tokens occurs in several different places, i.e., *global condition*. Examples for the first case are shown in Fig. 7.6(b). The conditional breakpoint attached to the place `Package` will stop execution only if more than one package is produced and the conditional breakpoint attached to the transition `Class2Table` stops execution if the variable `package` is bound to the token `p2`. An example for the latter case is shown by the breakpoint attached to the place `Table`, which stops execution if the `Table` place contains more than one element and if additionally the `Schema` place contains more than two elements.

7.2.2 Inspection

A natural prerequisite for reasoning about the state of execution is to provide appropriate inspection mechanisms. In the following it is shown, how the actual state of execution and the control flow is represented in Transformation Nets.

State inspection. Since Transformation Nets provide an integrated view on the transformation specification, i.e., not only the transformation logic itself is represented, but also the involved source and target metamodels as well as their according model elements. Thus, the

actual state of the transformation is presented to the transformation designer at any time during the transformation.

Visualization of control flow. As stated in Subsection 7.2.1.1, on the one hand, the visualization of the control flow is achieved by highlighting the transitions ready to fire. On the other hand, the history of transitions (which is hidden per default, but may be made explicit by the transformation designer) as well as the trace tokens provide visual information on which source tokens have been transformed to which target tokens (cf. Fig. 7.7). In order to make this information even more explicit, interrelationships between tokens are highlighted on mouse-over. For example, when moving the mouse over a source object token, the relationship to according value and link tokens that are contained by this object as well as already transformed tokens that originate from the source object token are highlighted by means of dashed lines (cf. Fig. 7.7).

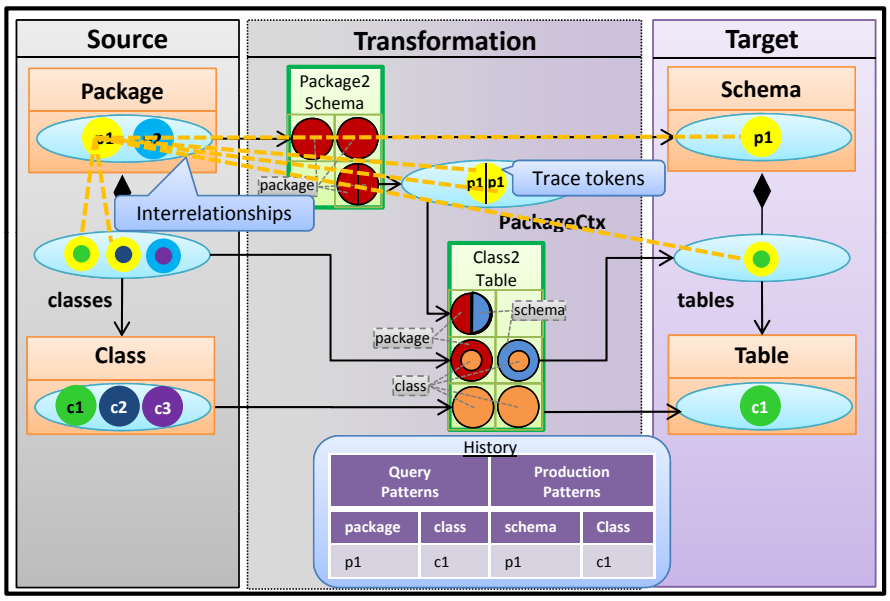


Figure 7.7: Visualization of Control Flow in Transformation Nets

7.2.3 Dynamics

A transformation designer may investigate the dynamic semantics of the transformation specification by a stepwise firing of transitions. Thus, it is possible to exactly follow which source element is transformed into which target element. Considering modules, the alternatives (i) *step into*, (ii) *step over*, (iii) *step return*, known from debugging in object oriented programming, are provided. Step into switches from the blackbox view into the whitebox view of the modules and fires an enabled transition. If more than one transition is enabled, one is chosen in a non-deterministic way. The semantics of step over is, that all contained and enabled transitions are fired. Please note that, if the firing of a transition enables another transition contained in the corresponding module, this transition is also fired. Finally, step return is enabled only in

the whitebox view of a module and means that all enabled transitions contained in the module should be fired and afterwards it should be returned to the blackbox view.

In summary, Transformation Nets support the features expected from debuggers, but rather on the model level than on the code level, as typically provided by current debuggers integrated in model transformation languages. Nevertheless, the explicit runtime model allows for more sophisticated means of debugging, i.e., to tackle the known problem of reasoning backwards in time as discussed in the following.

7.3 Query-Based Debugging

Debugging suffers from the well-known problem that programs execute forward in time whereas programmers must reason backwards in time to find the origin of a bug [173]. In this respect, the transformation designer needs to carefully approach the moment when the actual infection is observable in the transformation specification. Nevertheless, this needs not to be the exact point where the infection has been introduced, i.e., the actual defective piece of code. Thus, the transformation designer has to restart debugging and try to find the failure at some earlier point in time of execution. This is typically time-consuming and cumbersome and therefore means are needed that allow to reason backwards in time. Thereby, questions like "where does this target element come from" should be answered, i.e., *query-based debugging* should be enabled.

A first idea in this direction was the so-called *omniscient debugging* [120, 173], where debugging is based on a before recorded run of the execution. Nevertheless, the main disadvantage of this approach is that recording every single step is expensive in terms of memory and leads to time intensive operations during the actual debugging process. Ideas have been presented that tried to minimize the amount of the recorded data, i.e., it has been tried to abstract from details by means of representing the program execution in terms of runtime models, which is basically the idea of runtime verification [11]. Nevertheless, such a model on the execution is only the first step, but what is needed are means to analyze dependencies, i.e., which transition produced a certain token. The investigation of dependencies of a concrete program run is called *dynamic slicing* [173], i.e., deduce those parts of the transformation specification that deal with a certain element. In order to be able to calculate such dynamic slices, explicit trace information is needed. Since Transformation Nets provide explicit trace information, OCL queries can be employed to realize dynamic slicing, as discussed in the following.

7.3.1 Dynamic Slicing and Backwards Reasoning by Means of OCL

Since the execution of a transformation is stored as a model, which conforms to the Transformation Net metamodel (cf. Fig. 4.3 on 73), OCL queries may be employed to realize dynamic slicing for Transformation Nets in order to enable backwards debugging in time for model transformations. Fig. 7.8 shows the application thereof reusing the example already presented in Subsection 4.4.3. When inspecting the generated target model, at first sight it remains unclear, which transition and due to which configuration in the source model the column `a5` has been generated, since both transitions, `Attribute2Column` and `SuperAttr2Column` target the

place `Column`. The according token might be asked for the transition that created the token by means of the derived OCL function `getCreator()` which is depicted in Listing 7.6. Thereby, the function first gets its according place and collects the transition (cf. line 3). Afterwards, the history of the transitions is checked whether it contains the `id(s)` of the according token in its postconditions. The derived function `getIds` (cf. line 7) delivers the ids as set of the according token, i.e., `oid` in case of `Object` tokens, the `oid` and the `valueId` in case of `Value` tokens, `soid` and `toid` in case of `Link` tokens or `soids` and `toid` in case of `Trace` tokens.

Listing 7.6: GetCreator Function

```
1 context Token: def getCreator(): Transition =
2 -- collect all transitions targeting the token's place
3 self.place.inArcs->collect(a : TPArc | a.source.transition)
4   -- select the history
5   ->select(historyEntries
6   --check if the postcondition contains the according id of the token
7   ->collect(postcondition)->includesAll(self.getIds()))
```

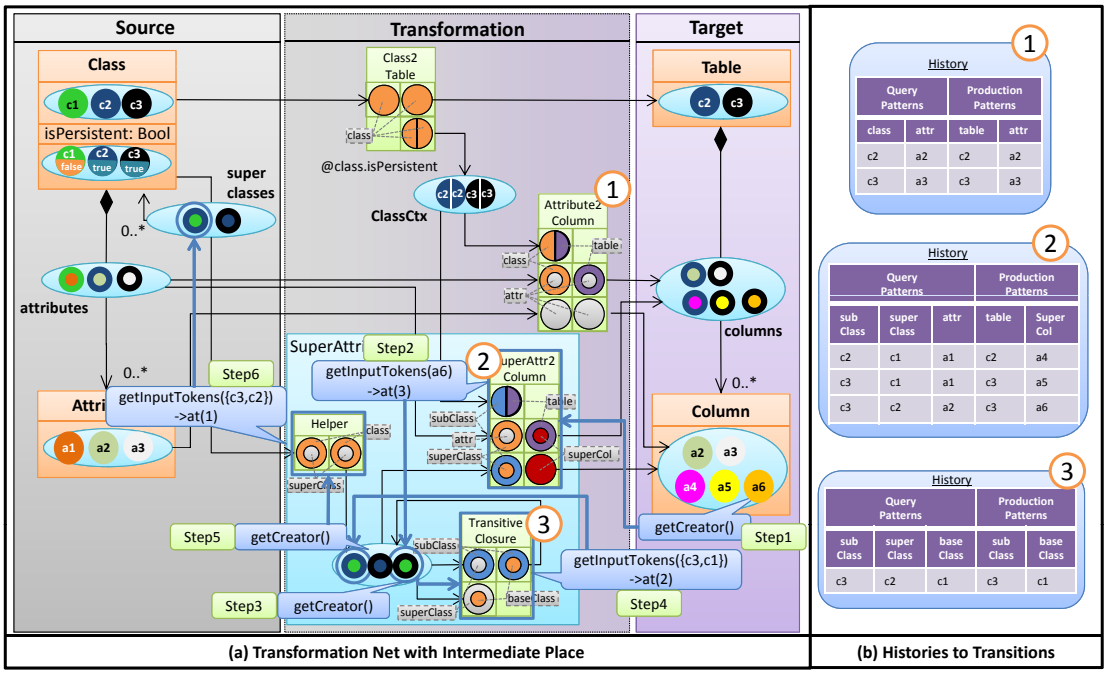


Figure 7.8: Backwards in Time Reasoning in Transformation Nets

In the case of the example, the histories of the transition `Attribute2Column` and `SuperAttr2Column` are investigated. As can be seen in Fig. 7.8, only the history of the transition `SuperAttr2Column` contains the token `a5` in the history's production tokens and therefore this transition is returned as according creator transition. This transition may then be asked for the binding of the according created token, e.g., the transition `SuperAttr2Column` might be asked for its query patterns by employing the OCL function `getInputTokens(t)`, whereby the parameter `t` denotes an according generated token, e.g., the token `a5` in our example. The

according OCL expression is shown in Listing 7.7, whereby in line 3 first all potential source tokens are collected. From these source tokens, only those tokens which are in the precondition of the history of token `t` are selected. In line 7, it is first checked if a postconditions contains the ids from the token `t`, and if this is the case, then the precondition is selected. Only if this precondition contains the `ids` of a certain token (iterator variable `x`), this token is added to the sequence to return.

Listing 7.7: GetInputTokens Function

```
1 context Transition: def getInputTokens(t: Token): Sequence{Token} =
2 -- select tokens of the source places, whose id is in the precondition of a history
3 self->collect(queryPatterns).inArc.source.tokens()->flatten()
4     --select those tokens, that are in the history of the transition
5     ->select(x:Token | (self->collect(historyEntries)
6     --get the precondition if the postcondition contains the id of token t
7     ->flatten()->select(h: History | h.postcondition->
8     includesAll(t.getIds())).precondition)->includesAll(x.getIds()))->asSequence()
```

In our example, the `getInputTokens` functions would return the context token `c3|c3`, the link token having the source Class object `c2` and the target Attribute object `a2`, as well a link token having as source object `c3` (cf. value of variable `subClass`) and as target object `c1`. Since there is no source link from `c3` to `c1` available in the source model (cf. link tokens in place `superclasses`), this token is further investigated. For this, the third token in the sequence is selected. On this token again the function `getCreator()` is called, which delivers the only available transition `TransitiveClosure`, since its history contains the according markings (cf. ③ in Fig. 7.8). It may be seen that an additional link `c3` is created (which is bound to the variable `subClass`) and `c1` (which is bound the variable `baseClass`) since there exists a link between `c3` and `c2` as well as `c2` and `c1`. In this respect, the transitive inheritance relationship between `c3` and `c1` is made explicit. It is then possible to ask for the according input tokens and their according creator again until the according tokens in the source model may be found (cf. steps 4 to 6 in Fig. 7.8).

7.3.2 Forensic Debugging

As stated in [66], the detailed traceability available in numerous of today's model transformation languages would allow for forensic debugging, i.e., debugging of a model transformation after its actual execution. In this respect, at least parts of the debugging task could be automated in order to narrow the scope of the potential location of the defect. For this, query-based debugging may again be incorporated. In [66], failures have mainly been divided into failures that lead to *invalid output*, i.e., the generated target model does not conform to its corresponding metamodel, or failures that lead to *incorrect output*, i.e., logical errors, which could be detected by means of forensic debugging.

Invalid output may be detected in Transformation Nets by investigating the tokens in the target places, e.g., dangling references, as already described in Section 7.1. Nevertheless, this is only where the failure shows its effect but again, support is needed to detect the actual origin. In this respect, query-based debugging mechanisms may be employed to reason backwards in time. For detecting *incorrect output*, in [66] only informal debugging questions, e.g., "Why are there no objects of type t in the target?" are presented, although the authors state that oracles could

be used for this purpose. However, a method called *re-enactment* is proposed which allows the "selective re-execution of logical parts of the model transformation in a controlled runtime environment to gather knowledge about specific problems" [66].

To realize re-enactment in Transformation Nets, first PaMoMo contracts (cf. Chapter 3) may be used to check the correctness of a transformation. As a simple example, two contracts are specified in Fig. 7.9. The first contract checks if there exists an equally named `Schema` instance for every `Package` instance and the second one checks if for every persistent `Class` there exists an equally named `Table`. When testing the specified QVT Relation specification (cf. specification in the middle of Fig. 7.9) against the contracts, it may be seen that the second contract fails, revealing which source elements caused the contract to fail (cf. verification results

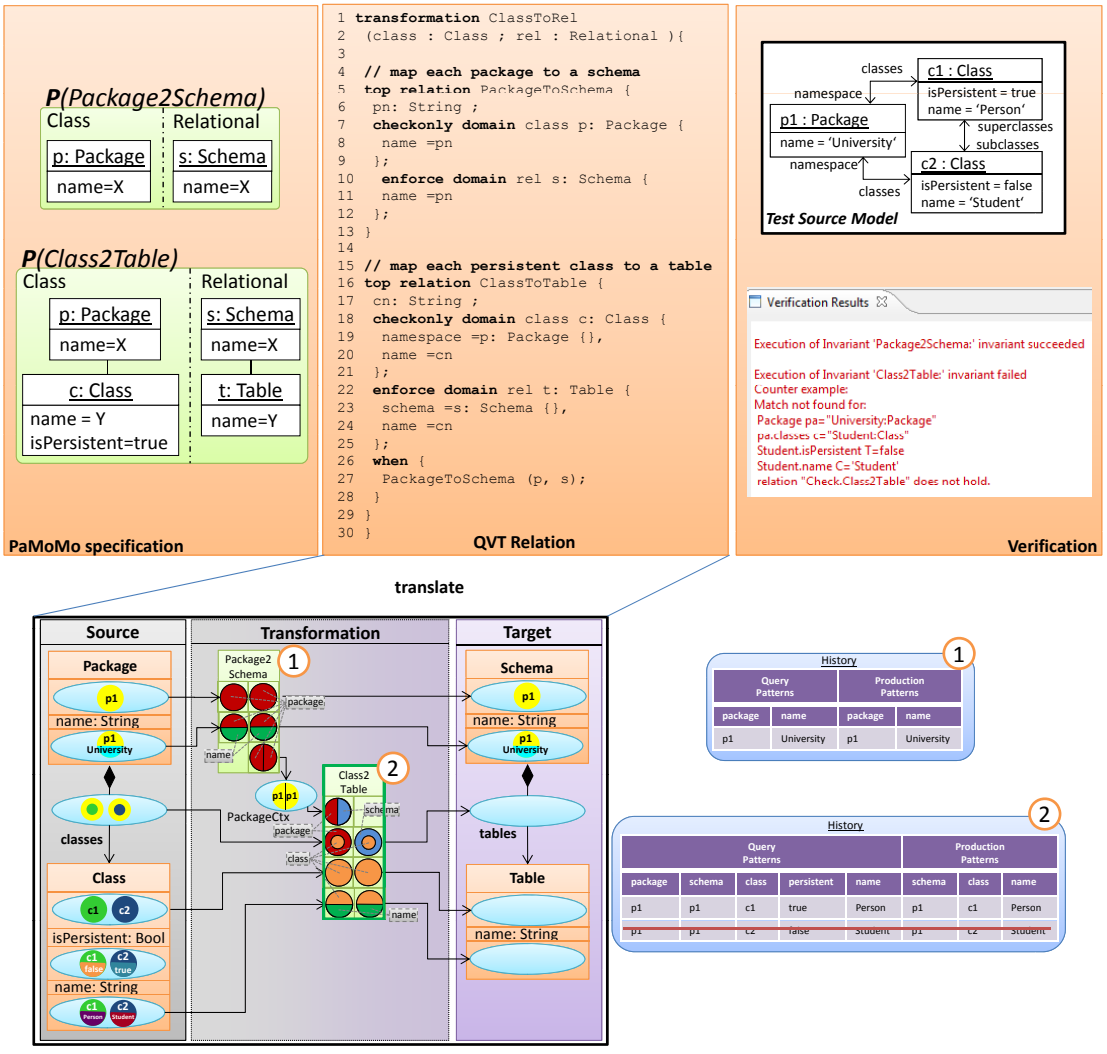


Figure 7.9: Re-Enactment: Combining PaMoMo and Transformation Nets for Debugging

in Fig. 7.9). Therefore, transitions that transformed these source instances need to be made available for re-enactment, i.e., it should be possible to fire them again to check for failures. Consequently, the precondition of the transitions' histories are investigated by searching for source tokens that caused the contract to fail. In order to be able to fire this transition again, the according history entries need to be deleted (cf. history ② in Fig. 7.9). Furthermore, the histories of all dependent transitions are cleared as well in order to consider the dependent transitions for re-enactment. In this respect, it is possible to identify those parts of the Transformation Net that were likely to cause a failure. Nevertheless, it is not possible to identify the exact point where the faliure has been introduced, for which again the afore mentioned (live) debugging features may be employed. When investigating the potential bindings of the transition `Class2Table` in the example, it may be detected that both classes `c1` and `c2` represent valid bindings although only the class `c1` is persistent. Therefore, the corresponding relation `Class2Table` of the QVT Relation specification misses an according condition that checks for persistent classes.

In summary, Transformation Nets provide means for backwards in time debugging by employing predefined OCL functions. Additionally, custom OCL functions may be used, since the execution of a model transformation is again represented as model, i.e., the runtime model may be queried by any OCL expression, which allows for flexible means of query-based backwards in time debugging. Backwards in time debugging is especially useful when oracles for testing the transformations are employed. The differences between the actually generated and the desired target model may be made explicit in the Transformation Net, i.e., if too many tokens are created, these tokens may be highlighted since they may serve as a starting point for debugging backwards in time.

7.4 Property-Based Debugging

Transformation Nets form a DSL on top of CPNs and the formal properties of CPNs may be applied during debugging, allowing for *property-based debugging*. For this, the state space of the CPN has to be constructed to calculate diverse behavioral properties. In a first step an overview on the state space analysis methods is given, before the properties and their applicability in the domain of model transformations are discussed in detail.

7.4.1 Calculation of the State Space

The basic idea of state spaces is to calculate all reachable states (markings) and state changes (occurring binding elements) of the CPN. In the resulting directed graph, nodes correspond to the set of reachable markings and the arcs correspond to occurring binding elements. Consequently, a state space depends on a specific initial marking, i.e., the according source model in Transformation Nets. State spaces provide powerful means to analyze the specified CPN and may be created fully automatically. The construction of the state space starts with the initial marking of a CPN. To exemplify this, Fig. 7.10 shows a simple Transformation Net, its according CPN, as well as the calculated state space. Thereby, the state labeled `1` represents the initial marking, which may also be seen from the fact that this is the only state without a predecessor. When examining the transformation logic, it may be detected that the transition `Package2Schema`

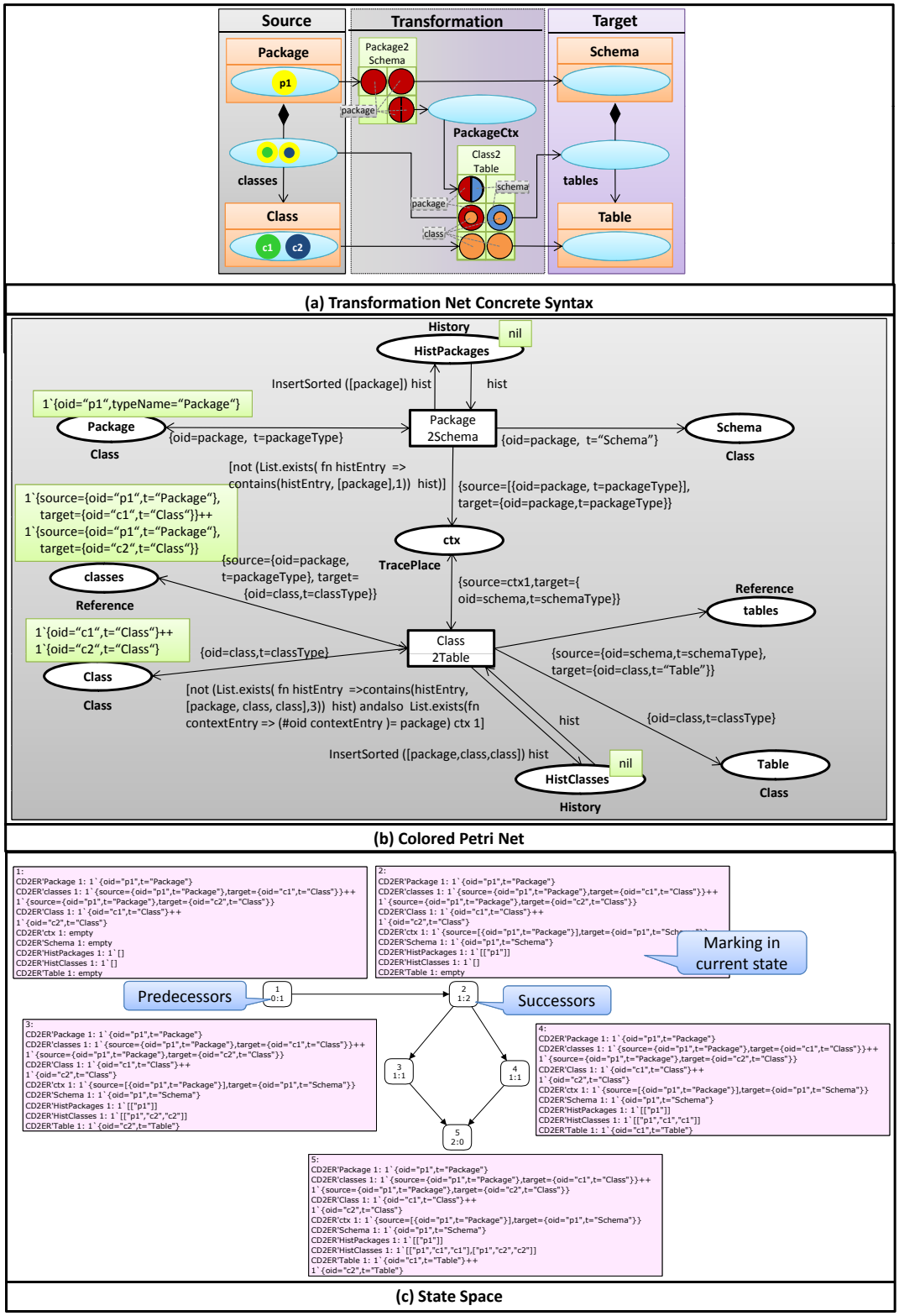


Figure 7.10: State Space of an Exemplary Transformation Net

can fire once, since there is only a single Package `p1` available (represented by the state number `2` in Fig. 7.10(c)). When inspecting this state, the CPN engine is allowed to chose either one of the two possible bindings, i.e., it may choose non-deterministically either the binding `{package=p1, class=c1}` or the binding `{package=p1, class=c2}`. The according state in the state space exhibits two successors. As might be already imagined by this simple example, the main drawback of calculating the state space is the so-called state space explosion problem [155], i.e., the size of the state space gets too large to be stored in memory. Nevertheless, since test input models may be assumed to be rather small, property-based debugging is a powerful mechanism to test and debug model transformations. A discussion about the state space explosion problem is considered in more detail in Chapter 9.

7.4.2 Behavioral Properties for Debugging Model Transformations

The calculated state space is used to calculate general properties on model transformations, e.g, termination or confluence. In the following, it is shown how such properties (cf. [112] for an overview) may be used to enable verification-based debugging of model transformation, as depicted in Fig. 7.11. Please note that the calculated properties base on the calculated state space and are therefore dependent on the actual source input models.

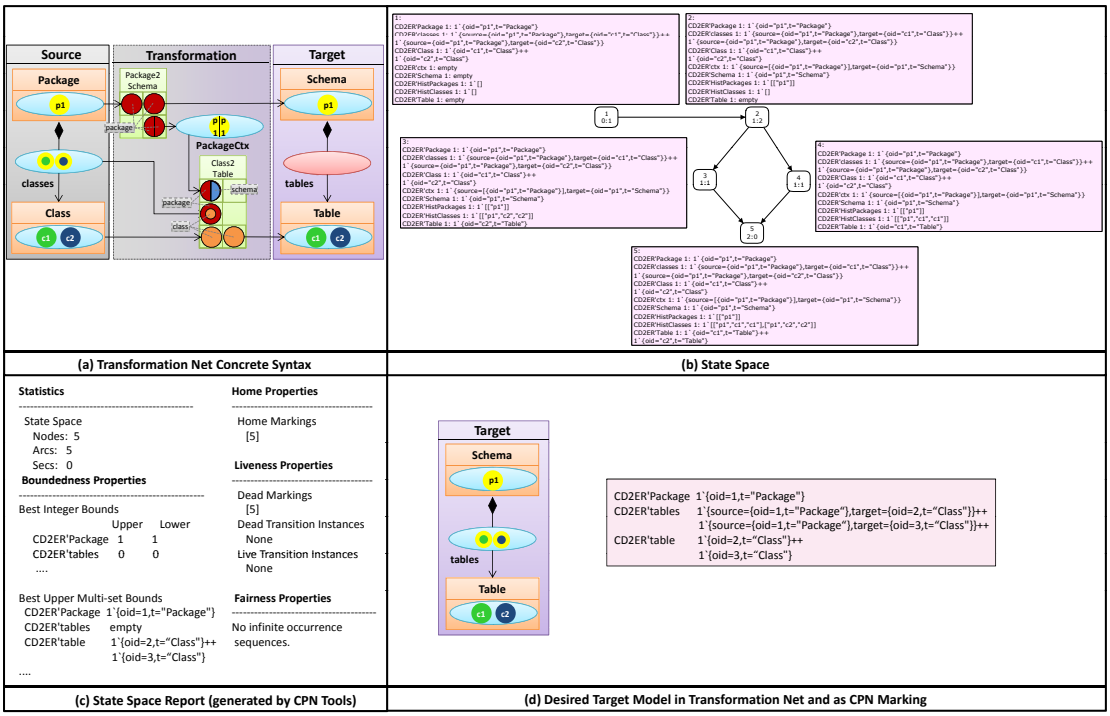


Figure 7.11: Application of CPN Properties for Debugging of Model Transformations

7.4.2.1 Termination and Confluence Verification using Dead and Home Markings

In a batch and exogenous model-to-model transformation scenario (which is the focus of this thesis), a model transformation is always required to terminate. Thus, the calculated state space needs to contain at least one *Dead Marking* [112], which is a state in the state space without any successors, i.e., $\exists M$ such that $M_0 \overset{\sigma}{\Rightarrow} M$ and $enabled(M) = \emptyset$, meaning that after a certain firing sequence, starting from the initial marking M_0, a marking M is reached, where no bindings are enabled any more (cf. state 5 in Fig. 7.11(b)). If such a dead marking is found, the specified transformation is considered to be partially correct, i.e., if execution terminates, a correct target model might be possible. Please note that although the introduced history concepts ensure that a transition may only fire once for a specific combination of input tokens, it may not ensure termination since if a transition occurs in a cycle, i.e., a place is both, a source and a target place of a single transition and if the transition produces new objects (cf. new colors in Transformation Nets) every time it fires, the history concept may not ensure termination. However, such cycles may be detected at design time and are automatically prevented for Transformation Nets[1]. Therefore, termination may be statically ensured in Transformation Nets, which is in contrast to model transformation languages based on graph grammars, where termination is undecidable in general [119]. In this respect, the state space report generated by CPN tools shows that there exists a dead state having the id 5 (cf. Fig. 7.11(c)).

Nevertheless, in order to formally verify termination, it has to be ensured that a dead marking is always reachable, i.e., in every possible execution. For this, *home properties* are provided in CPNs, whereby a *Home Marking* M_{Home} is a marking, which may be reached from any other reachable marking, i.e., $\forall M | M \overset{\sigma}{\Rightarrow} M_{Home}$ [112]. As stated in [72, p. 171], this means that "it is impossible to have an occurrence sequence starting from M_0 which may not be extended to reach M_{Home}". For the example, state 5 represents a home marking, as can be seen in Fig. 7.11(c). Consequently, if the state space contains a single *Dead Marking* which is equal to a single *Home Marking*, i.e., both states offer the same id, it is ensured that the CPN (and thus the according Transformation Net) always reaches a dead marking leading to a confluent CPN, i.e., there exists a unique terminal marking, that may always be reached. Formally this is denoted as *if* $\forall M, M' M_0 \overset{\sigma}{\Rightarrow} M \wedge enabled(M) = \emptyset \wedge M_0 \overset{\sigma}{\Rightarrow} M' \wedge enabled(M') = \emptyset$ *then* $M = M'$. Since the calculated properties depend on the actual source model, in general the transformation would have to be tested with all possible input elements. Thus the question arises, in which situations a non-confluent behavior may occur, i.e., when does a transformation contain more than one home or dead state. As discussed in Chapter 2, non-confluence in model transformations may occur if two rules are non-parallel independent [65]. The same is true for CPNs if the specified net is not *persistent*. A CPN is said to be persistent if "for any two enabled transitions, the firing of one transition will not disable the other one" [112], which is equal to the definition of parallel-independence. Consequently, this property has to be ensured for Transformation Nets, which is ensured by the fact that the source places are only accessed in a read-only manner. Furthermore, it is not allowed to specify conditions on trace information, which otherwise may lead to non-determinism, as well as to use negative patterns for intermediate and trace places. In general,

[1] Please note that recursion in general is allowed in Transformation Nets, but only prohibited for the the special case of new colors which would always lead to an endless recursion.

however, confluence may not be ensured statically and depends on the actual transformation logic, i.e., a check-before-enforce scenario with a non-unique key may lead to a non-confluent behavior (cf. Section 6.3 and Section 9.2). Since in the example above the state with the id 5 is a dead marking as well as a home state, this transformation may be considered as confluent (cf. Fig. 7.11(c)).

7.4.2.2 Model Comparison using Reachability and Boundedness Properties

To achieve a correct transformation result, an equal *Home Marking* and *Dead Marking* is a necessary, but not sufficient condition, as it may not be ensured that this marking represents the desired target model (which has to be decided by some complete oracle function). By exploring the constructed state space, it is possible to detect if a certain marking, i.e., the target marking derived from the desired target model, is reachable with the specified transformation logic. If this is the case, and if this marking is equal to both, *Home Marking* and *Dead Marking*, it is ensured that the desired target model may be created with the specified transformation logic in any case. Nevertheless, if we consider the example presented above, the derived marking from the desired target model (cf. Fig. 7.11(d)) is unreachable by the specified transformation, i.e., the marking may never be achieved.

If the desired target model is not reachable, a possible step to debug the transformation specification is to compare the target model generated by the transformation to an expected target model. To identify wrong or missing target elements in terms of tokens automatically, *Boundedness* properties [112] (*Integer Bounds* and *Multiset Bounds*) may be applied (cf. Fig. 7.11(c)). *Upper integer bounds* state how many tokens at most reside in a certain place, i.e., in a first step only the number of tokens may be compared. Since no tokens are consumed from a place, the number of tokens in a place representing a target metamodel element has to be equal to the number of tokens derived from a desired target model. On the one hand, if there are too few tokens, the according place is highlighted to give the transformation designer a hint for debugging. When deriving the upper integer bound of the desired target model, two tokens in place `tables` would be expected, but none is actually created (cf. Fig. 7.11(c)), and therefore the according place is highlighted in the Transformation Net (cf. Fig. 7.11(a)). Furthermore, such a situation might indicate that the according Transformation Net either specifies a too restrictive condition or that it misses according production patterns. On the other hand, if there are too many tokens, this might indicate that the specified condition is too weak. In this situation, the according tokens may be identified by using the *Multiset Bounds*, which contain the respective marking. These tokens are then highlighted in the according Transformation Net and the transformation designer might then make use of query based debugging mechanisms to actually discover the origin of the failure.

7.4.2.3 Transition Error Detection using Liveness Properties

As stated above, missing target tokens may be detected by boundedness properties, whereby one potential source of error might be a too strong condition on the according transition. Thereby, the situation might occur that a certain transition specifies a condition that is never fulfilled during the whole transformation process, i.e., the transition never fires. This situation may be detected

by means of so called *Dead Transition Instances* or *L0-Liveness* [112]. Dead transition instances may be found in the state space report, whereby none means that no transitions exists, which has never fired (cf. Fig. 7.11(a)). If a transition did not fire, it indicates that the source model did not enable the transition and therefore, either the specified test model did not consider a certain scenario, the specified transition is incorrect or even the source model is incorrect. In case of dead transition instances, the according Transformation Net transition gets highlighted in order to set the focus for debugging.

7.4.3 CPN Properties for Model Transformations

By applying and analyzing behavioral properties of CPNs it has been figured out which properties are useful in the context of model transformation testing and debugging and which kinds of errors may be detected. The proposed taxonomy (cf. Fig. 7.12) investigates possible locations of errors, classifies typical model transformations errors and shows, which properties are useful for their detection.

During specification of model transformations there are three possible locations of errors, either in (i) the metamodel, (ii) the model, or (iii) the transformation logic. The detection of errors in the metamodel is in general out of scope of transformation languages. As model elements are explicitly represented in Transformation Nets as tokens —in contrast to other transformation languages which typically do not represent the models — semantic errors in the model may be detected by liveness or boundedness properties. For example, an incorrect source model (e.g., a reflexive link represented by a link token with same inner and outer color) might lead to dead transition instances or an incorrect firing behavior of a transition and thus, to an incorrect number of tokens in the target place.

Errors in the transformation logic itself may be divided into errors local to a single transition (*Intra-Rule Error*) or errors which may only be detected by examining the interrelations between several transitions (*Inter-Rule Error*). Intra-rule errors may be divided into errors occurring at the precondition (LHS) or postcondition (RHS) of a transition. Common errors on both sides (e.g., a wrong matching pattern or a wrong instantiation of target models) may be detected by examining the boundedness properties in comparison to an expected target model or by custom state space functions checking if a certain marking is reachable. Due to the fact that these two properties may be applied in various scenarios special tool support is provided. Finally, dead transition instances point out that a given LHS specification of a transition may not be fulfilled by the given source model.

Inter-rule errors occur if transitions depend on other erroneous transitions or, if the specified transformation logic does not cover the whole source or target metamodel. Although these errors may be easily detected by checking for source places that have no arc to any transition or target places which are not target of any transitions (cf. Section 7.1), it is also possible to apply boundedness and reachability properties to detect these kind of errors. To verify if several transitions interact correctly, the home state and the dead state property may be checked. Additionally, the persistence property has to be fulfilled (which is statically ensured in Transformation Nets due to the non-consuming firing behavior). If the CPN is persistent and if there exists a single, equal home and dead state, then the specified transformation logic is confluent with respect to the test input model.

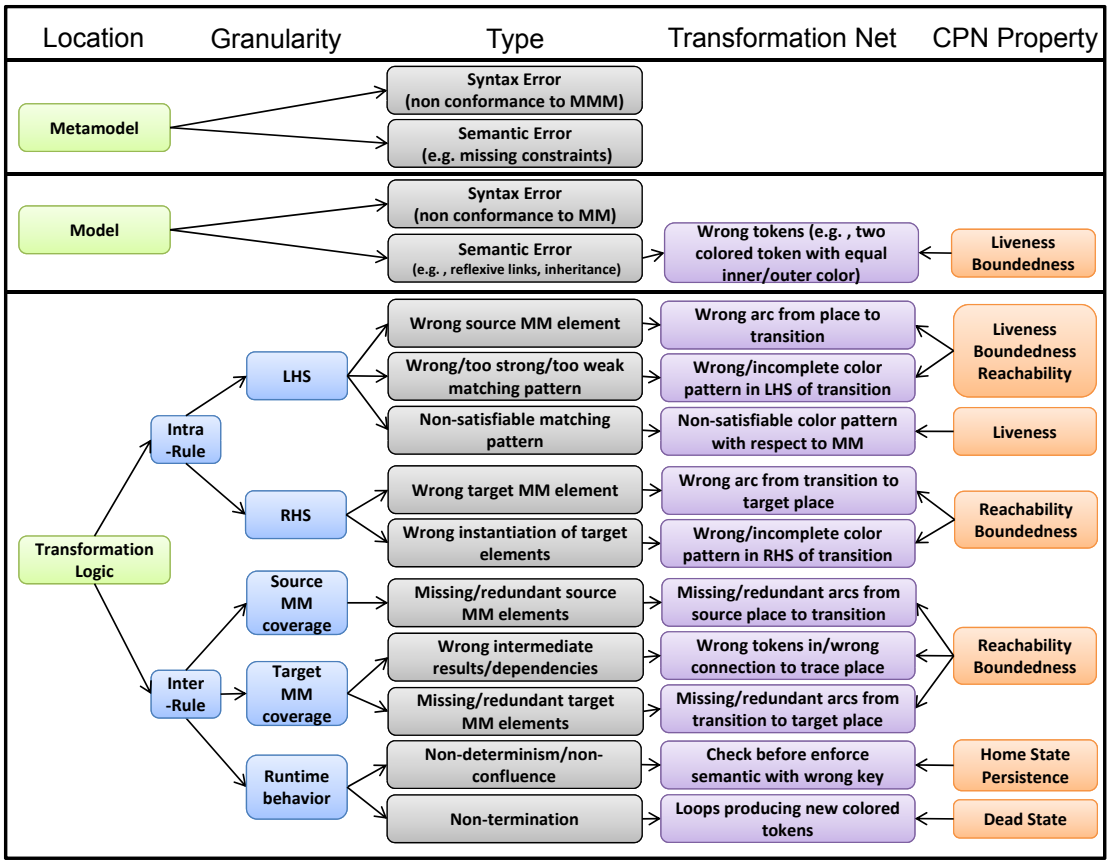


Figure 7.12: Taxonomy of Transformation Errors and CPN Properties

7.4.4 Towards Model Checking of Model Transformations

A limitation of behaviorial properties is that they heavily depend on the initial marking of the CPN, i.e., this means for model transformation that the properties may only be ensured to hold for a specific input model. Nevertheless, it is often demanded to formulate more general properties. Although PaMoMo contracts may be used for this scenario, as discussed in Chapter 3, PaMoMo does not allow to check properties concerning the actual execution of the model transformation, but only allows to check for valid source-target correspondences. For example, in PaMoMo it is not possible to check if a certain transformation rule that translates classes to tables has been executed as many times as classes are available. Theoretically, the tables might result from other rules, which is not desired. For this, reachability analysis in CPN Tools may be used in an automated way by means of a CTL-like logic called ASK-CTL [35]. ASK-CTL allows to query if a certain state or a state change (e.g. the occurrence of certain transitions) occurs sometimes or always in a certain path of the state space. For instance, the command `eval_node` *INV(POS(NF("Schema for Package", schemaForPackage))) InitNode*, which returns `false` if not for every schema a package is created by a certain transition, allows to

check execution specific properties. In the previous command, `InitNode` is the initial marking, `schemaForPackage` is a user-defined function that checks, if for every class a package has been created (by comparing the markings) and if the target markings originate from a certain transition. `POS(A)` demands the property `A` to be eventually satisfied, and `INV(A)` demands `A` to be satisfied in all possible paths. Nevertheless, as this simple example reveals, the specification of such functions is tedious and error-prone for a transformation designer, not being familiar with ASK-CTL. Furthermore, Transformation Nets currently do not provide means to specify ASK-CTL formula and the underlying CPN is hidden from the transformation designer (although it could be made explicit). Further research is needed to provide transformation-designer friendly means to specify such properties on the level of Transformation Nets. Nevertheless, this is out of the scope of this thesis and is considered as future work (cf. Chapter 10).

7.5 Fixing Failures

The last phase in the debugging process is to actually correct the defect. In Transformation Nets, the transformation designer is allowed to (i) alter the according model, i.e., tokens may be added, edited or removed, and (ii) to change the specified transformation logic, which is discussed in the following.

7.5.1 Adapting the Model

Since in Transformation Nets the model is explicitly represented by means of tokens, it should be possible to add, edit or delete certain tokens in order to fix a defect during debugging. Adding tokens to places is only allowed, if the according place is either a place representing a source metamodel element, a trace place or an intermediate place. Adding tokens in trace or intermediate places may be useful in order to continue debugging if the transformation unexpectedly terminated. By this questions like “would this transition fire, if there would be an according trace token” may easily be answered. Editing or deleting tokens is more complex than adding tokens, since those tokens might already have been matched by transitions. Thus, it may be the case that a transition might not have fired, if the token was not present or exhibited some different value. If a token is changed or deleted, the histories of the transitions have to be updated. Please note that in both cases – either editing or deleting – the according configuration is deleted from the transition’s history in order to allow to re-execute transitions depending on an edited or deleted token. Consequently, also the accordingly produced tokens and also the history of dependent transitions have to be updated. Finally, changes or deletions might also lead to dependent changes, i.e., if an object token is deleted all dependent value and link tokens are deleted as well. To exemplify this, Fig. 7.13 shows a simple example of how the tokens and the histories have to be updated when the owning object of a link is changed, e.g, the link from package `p1` to class `c3` is changed to a link from package `p2` to class `c3`. In a first step the history of those transitions are investigated, that query the reference place `classes` which is the transition `Class2Table` in the example. Consequently, those history entries that contain the old values of the token in the according variables, e.g., `package=p1` and `class=c3` in the example, are deleted. Additionally, the produced tokens are deleted as well, e.g., the link

token from package `p1` to class `c3` and the object token `c3` in the `Table` place are deleted (cf. Fig. 7.13(b)). Therefore, the transition `Class2Table` is again enabled and the transformation designer might debug the changed configuration.

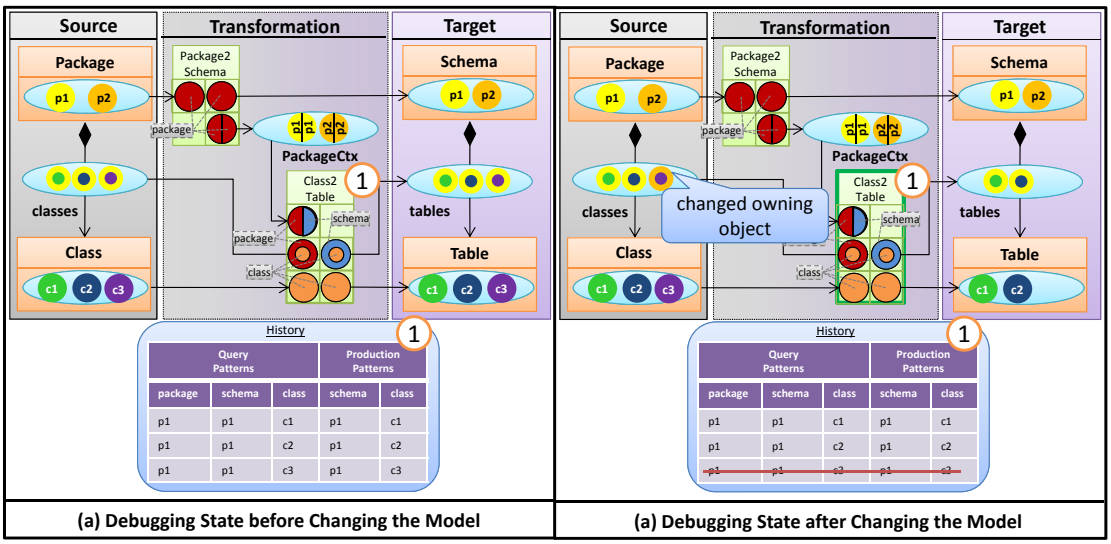


Figure 7.13: Changing the Model during Debugging

Finally, a remaining question is, if the changes in the model should be local to the debugging environment, i.e., the according changes in the source model should not be made persistent, or if the changes during debugging should be made persistent. On the one hand, if the changes are local to the debugging environment, the transformation designer is allowed to "play around" with certain model configurations without changing the according test input model. On the other hand, according changes in the debugging environment have to be probably repeated in the test input model, if errors have been detected in the model. Therefore, in Transformation Nets the changes are local to the debugging environment per default, but the transformation designer may explicitly commit the changes in order to persist them.

7.5.2 Adapting the Transformation Logic

In general, the transformation logic represented in Transformation Nets may be changed in a similar way as the models. In this respect, it is allowed to add transitions, trace or intermediate places during debugging. Furthermore, it is allowed to edit existing transitions, e.g., by adding further query tokens or deleting a production token. In any of these cases, the history of the according transition as well as those of dependent transitions have to be updated, in order to allow to re-evaluate the according parts of the transformation specification. Nevertheless, since Transformation Nets intend to provide a runtime model for declarative model-to-model transformation languages, i.e., it is possible to represent the actual transformation logic in terms of Transformation Net concepts, the back propagation of changes in the transformation logic to the actual transformation languages represents a major challenge. Currently, the back propaga-

tion requires the specification of an explicit transformation, i.e., not only a transformation from the transformation language to Transformation Nets is required, but also a transformation from Transformation Net concepts to the concepts of a specific transformation language. Thus, only if such a transformation is provided, the transformation designer might commit changes in the transformation logic in order to represent the changes automatically in the according transformation language. Nevertheless, a more generic back propagation, i.e., by means of an explicit trace model derived from the forward translation to Transformation Nets, together with a model representing specifics of a certain model transformation language might allow for an automatic backwards translation. However, this is not in the scope of this thesis and considered as future work (cf. Chapter 10).

In summary, on the one hand, Transformation Nets allow the transformation designer to flexibly change the according source model. On the other hand, although transformation logic may be changed, an explicit backwards transformation has to be provided in order to represent the changes in the according transformation languages, which will be discussed in more detail by means of QVT Relations in the case study in Section 9.2.

7.6 Summary

This chapter presented how the dedicated runtime model may be used for debugging model-to-model transformations. In a first step, the detection of code-smells has been discussed, i.e., it was shown that certain structures in Transformation Nets indicate potential sources of defects. Second, means for simulation-based debugging were presented. The transformation designer is enabled to execute the transformation stepwise, allowing to investigate the actual operational semantics. Thereby, means already known from common debuggers of programming languages have been adapted to the runtime model (e.g., breakpoints), but also model transformation-specific concerns have been discussed, e.g., how the non-deterministic selection of rules and the matching of source elements may be made explicit to the transformation designer. Besides means for live-debugging also means to tackle the well-known problem of reasoning backwards in time during debugging have been discussed. Predefined OCL queries allow to investigate the actual execution of a model transformation, e.g., to check which configuration of source tokens produced a certain target token. Furthermore, re-enactment has been discussed as a means to realize forensic debugging and to combine the results of PaMoMo contracts with the debugging features of Transformation Nets. Since Transformation Nets base on CPNs, it was shown how their formal properties may be used for debugging. In this respect, general properties were presented, but it was also shown how reachability analysis may be used for debugging. Finally, means for fixing bugs in Transformation Nets were discussed.

Up to now, the presented concepts have been discussed on a conceptual level, only. Therefore, the following chapter presents a prototypical implementation thereof, which is then used to prove the applicability of the presented concepts in the subsequent evaluation chapter.

Chapter 8

Prototype Implementation

Computers are useless. They can only give you answers.

— Pablo Picasso

Contents

The previous chapters introduced PaMoMo, a language for specifying model transformation contracts, and Transformation Nets as a runtime model for model-to-model transformation languages as well as its compilation into CPNs as a semantic domain. Thereby, the focus was on a conceptual level rather than on implementation details. Therefore, this section shortly elaborates on the prototypical implementations serving as a proof of concepts. First the tool PACO-CHECKER (short for PaMoMo Contract Checker) is presented which allows to graphically specify PaMoMo contracts, as well as their automatic compilation to QVT Relations in order to test an existing model transformation. Second, the DEBUT (short for Debugger for Transformations) tool gives its debut, which provides a graphical editor for the runtime model. Finally, the compilation of Transformation Nets to CPNs is discussed as well as how the CPN Tools environment can be used for simulation and state space analysis.

8.1 PaCo-Checker - PaMoMo Contract-Checker

In order to verify whether a transformation logic fulfills the requirements specified by PAMOMO contracts, the patterns are compiled to QVT Relations. For automating this process as well

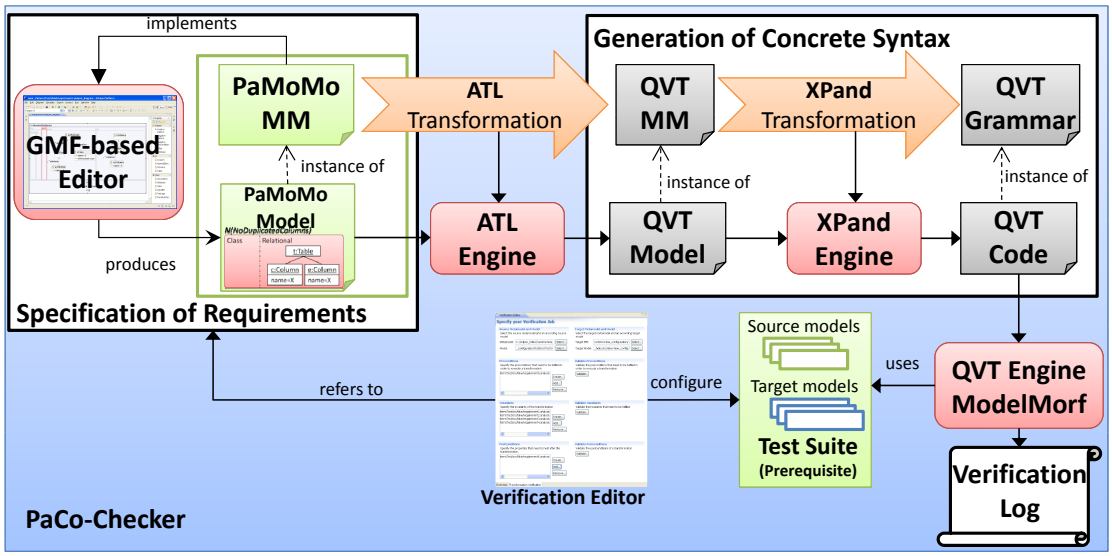


Figure 8.1: Overview of the Architecture of PACO-Checker

as for the visual specification of contracts, an EMF-based tool, called PACO-Checker has been prototypically developed, whereby its architecture is shown in Fig. 8.1. This section provides in the following an overview on the needed steps for the verification process, presents the different components of the tool, and illustrates its use.

8.1.1 Prerequisites.

For using PACO-Checker, the existence of the source and target metamodels is a prerequisite, as these are necessary to specify the contracts and to implement the transformation. In addition, for the verification process, a suitable set of input models conforming to the source metamodel is needed. Such input models can be manually created, which however is a tedious and error-prone task, leading to small input models that cover parts of the metamodel only. Alternatively, there are mechanisms available that automatically synthesize a large number of different input models [23, 46, 140] ensuring a certain level of metamodel coverage, as already discussed in Section 2.1. The existence of such a set of input models is assumed as well, since their generation is out of scope of this thesis.

8.1.2 Formal Specification of Requirements with PAMOMO.

In a first step, the transformation requirements have to be formally specified using PAMOMO. For this purpose, PACO-Checker provides an implementation of the PaMoMo metamodel (cf. Fig. 9.1) using EMF. Based on the implementation of the metamodel, PACO-Checker provides a graphical concrete syntax supported by a GMF-based [54] editor, which enables the visual specification of PaMoMo contracts (cf. Fig. 8.2). In order to be able to use elements of the source and target metamodels involved in the transformation, these have to be imported into the tool palette of the editor before starting modeling the patterns. Afterwards, the transformation designer can

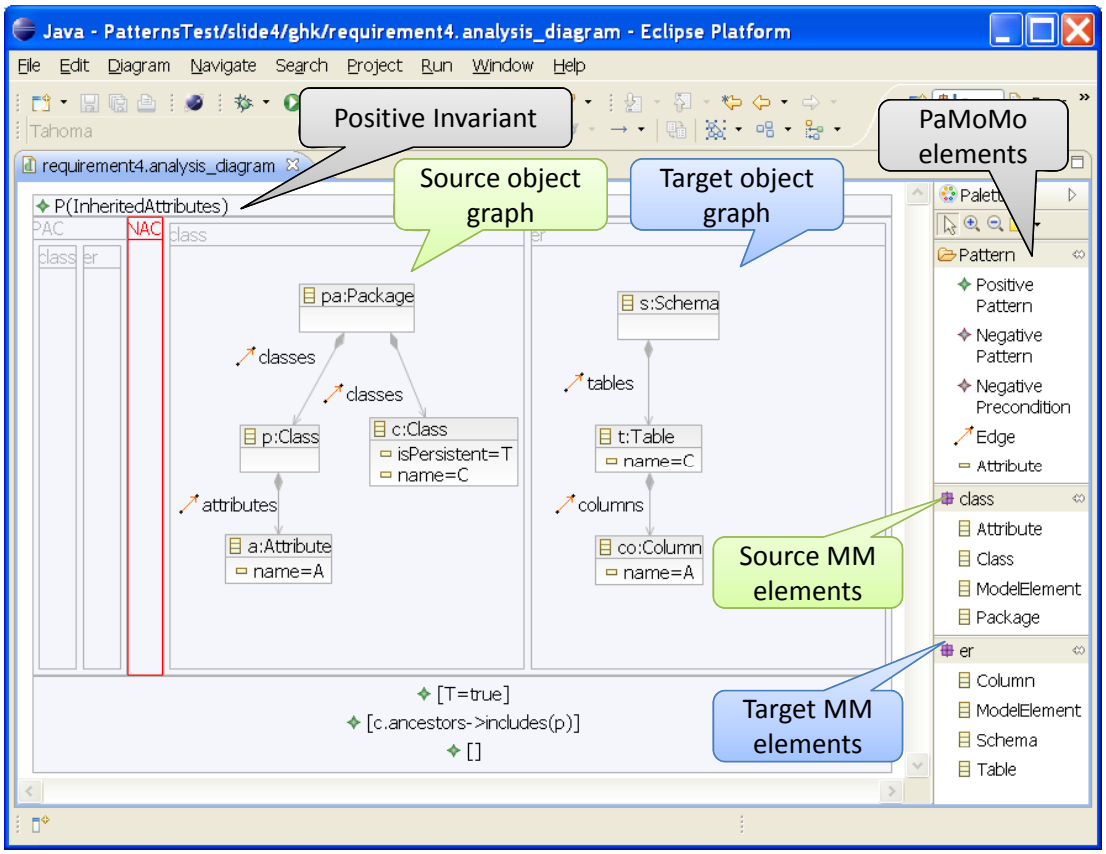


Figure 8.2: Specification of Invariant for *Requirement 4* (cf. Fig. 3.6) with PACO-Checker

use the editor to specify preconditions, postconditions and invariants forming the desired model transformation contracts. The current implementation supports one type of pattern per contract only (whereby one contract results in one file), i.e., in a contract the transformation designer either specifies preconditions, postconditions or invariants. Therefore, if preconditions, postconditions and invariants should be used to verify a transformation, three different contract files are needed. Fig. 8.2 shows a screenshot of the user interface whereby an invariant is modeled, representing requirement 4 of the running example (cf. Section 1.2). As can be seen, classes from the source or target model can be put into the according compartments of the invariant. Additionally, the features of the according classes may be added. In order to specify conditions on the attributes, e.g., to check if the class is persistent (boolean attribute `isPersistent`) or more general conditions, e.g., if class `p` is a superclass of class `c`, according OCL expressions may be specified at the bottom of the pattern.

8.1.3 Specification of a Verification Job.

Once the designer has specified the contracts, a *verification job* has to be configured. Such a job definition allows executing all specified preconditions, postconditions, and invariants to achieve a comprehensive verification result. Fig. 8.3 shows a screenshot of the job specification for the

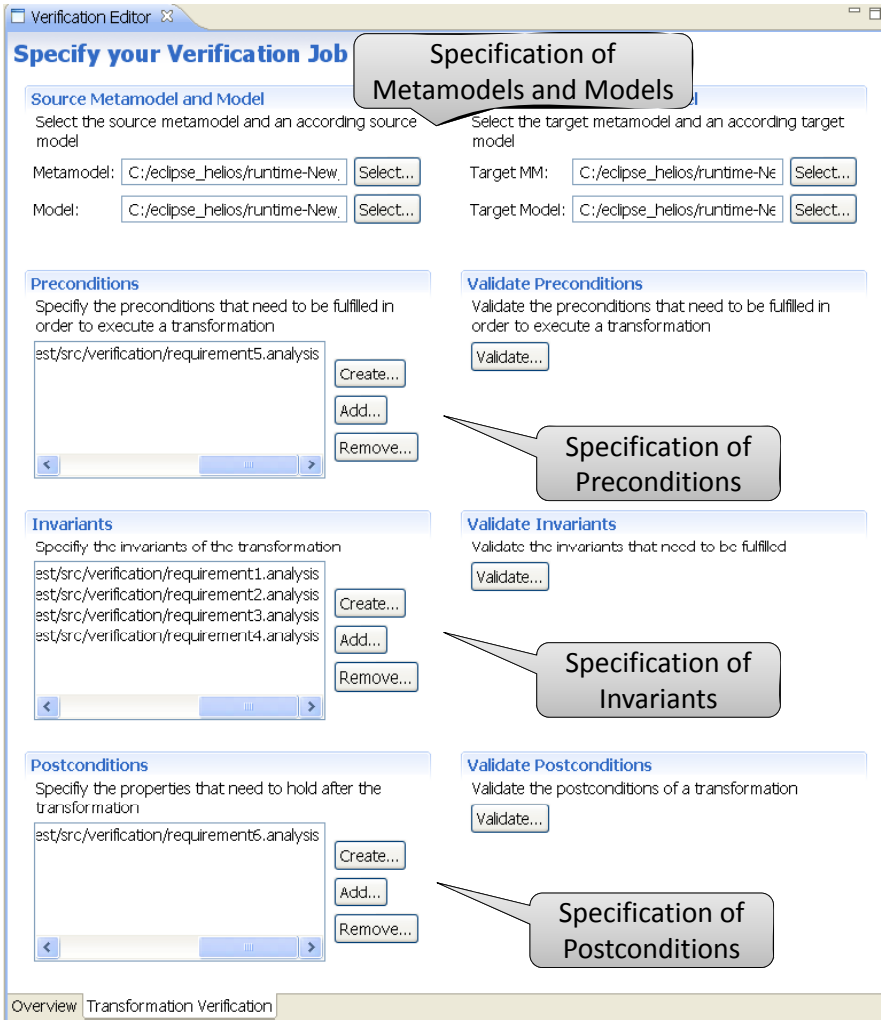


Figure 8.3: Definition of a Verification Job with PACO-Checker

running example. In a first step, the source and the target metamodel have to defined as well as their corresponding models. In this respect a source (test) input model is needed as well as the target model generated by the transformation specification under test. In a second step, the contracts which shall be checked for the specified transformation have to be assigned, i.e., preconditions, postconditions, and invariants may be added. Thus it is possible to reuse patterns to verify several transformations, e.g., if the source model should be transformed into several target models by different transformations, the preconditions may be reused.

8.1.4 Execution of the Verification Job.

Once specified, the verification job can be executed if no inconsistency between the patterns of the contract is reported by the reasoning component. In order to execute the job, an ATL trans-

formation transforms the PAMOMO contract into a QVT model implementing the semantics of the contract as described in Section 3.5. Since there is no execution engine available to execute QVT Relations on the basis of its abstract syntax, the QVT concrete, textual syntax has to be produced by means of a model-to-text transformation. For this, PACO-checker makes use of the template-based language XPand[1]. The resulting QVT Relations code is finally executed by the ModelMorf QVT Relation engine [149] in checkonly mode. The QVT Relation engine produces the verification log, providing hints of any error in the transformation logic.

8.1.5 Inspection of Verification Results.

After executing the verification job, the tool produces a verification log. Fig. 8.4 shows to the right the log generated for the running example, considering the input and output models to the left. The actual output model was generated by executing the QVT Relations specification depicted in Fig. 3.18. This log reports that requirements 1-3 are satisfied for these models, but not requirement 4, which addresses the translation of inherited attributes (as already discussed in Section 3.6).

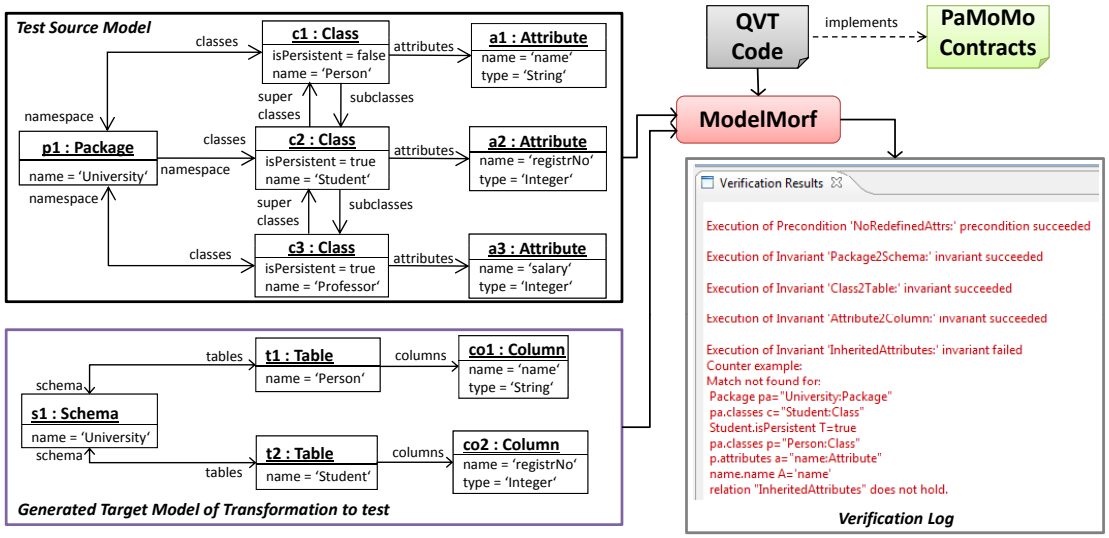


Figure 8.4: Verification Results of Requirements 1-4 for the Running Example

Additionally, the verification log may be serialized into a *verification model*, which may then serve as input for the debugging process (cf. Section 8.2). The according metamodel is depicted in Fig. 8.5. Thereby a `VerificationProject` holds references the the involved source and target `Metamodels` which again hold a reference to the currently used source and target `Models`. Furthermore, a `VerificationProject` stores the `Patterns` that should be verified. For this, different roles are provided, cf. `VerificationProject.preConditions`, `VerificationProject.invariants`, and `VerificationProject.postConditions`. For every pattern an `ErrorTrace` may be stored in case the `Pattern` failed. In

[1] http://wiki.eclipse.org/Xpand

order to allow for re-enactment of the transformation during debugging (cf. below) also a reference to the according `DomainPattern` stemming from the QVT Relations metamodels is provided, i.e., it is possible the store the according bindings.

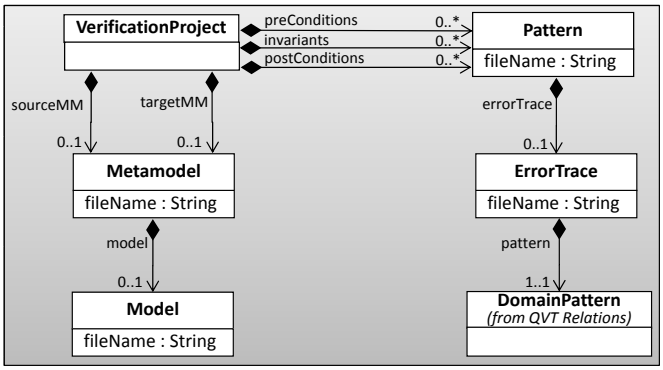


Figure 8.5: Metamodel of Verification Log

8.2 DEBUT - DEBUgger for Transformations

After discussing the prototypical implementation of the PACO-Checker tool for the specification and execution of PAMOMO contracts, this section elaborates on the DEBUT prototype, which implements the proposed runtime model. Furthermore, it builds the basis for the debugging features, whereby their implementation is discussed afterwards.

8.2.1 Overview on Debut

The DEBUT tool consists of several components in the form of Eclipse plugins, which themselves base on the infrastructure provided by the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF). The component diagram depicted in Fig. 8.6 gives an overview of the provided plugins and their interdependencies (dependencies to base plugins and external plugins are not shown).

Transformationnet.model. The central point of DEBUT is the realization of the runtime model, whose implementation is encapsulated in the `transformationnet.model` plugin. Thereby the implementation of the metamodel follows the metamodel presented in Fig. 5.5. Additionally, the proposed OCL invariants, e.g., to check inheritance constraints as well as the constraints presented in Section 7.1, have been accordingly implemented.

Transformationnet.edit. These plugin serves as a controller between the model and the clients that want to edit a Transformation Net and is generated by the EMF infrastructure.

Transformationnet.diagram. In order to provide a concrete visual syntax, a GMF based visual editor has been implemented that allows to visualize and edit according Transformation Nets. A screenshot thereof is depicted in Fig. 8.7.

Transformationnet.diagram.ui. This plugin provides user interface components in order to conveniently create or edit Transformation Nets, e.g., custom property sheets, as well as dialogs and wizards to edit a Transformation Net.

Transformationnet.debugging. The debugging features of the Transformation Net are encapsulated within this plugin, e.g., the definition of breakpoints and the predefined OCL expressions for debugging backwards in time.

Transformationnet.debugging.ui. In order to separate the debugging logic from its concrete visualization, this plugin provides the necessary user interface components for the debugging process, e.g., context menu entries or user interface components for the visualization of the history.

Transformationnet.adapter. This plugin provides a basic infrastructure in order to be able to load source and target metamodels and models from existing files. In this respect, an extension point is provided, that allows to implement a specific adapter for a certain meta-metamodel. The prototype provides adapters for Ecore, XML and OWL (encapsulated in according plugins, as can be seen in Fig. 8.6). Additional meta-metamodels may be supported by providing a specific implementation of the extension point.

Transformationnet.bridge. In order to be able to debug transformation logics specified in declarative model-to-model transformation languages, bridges between the according transformation language and the Transformation Net formalism are needed, i.e., transformation specifications in a certain transformation language should be represented in the Transformation Net formalism. In this respect these plugins encapsulate commonalities between bridges to different transformation languages and specify an extension point to ease the integration of bridges to specific transformation languages. In the current prototype, bridges for QVT Relations and for Mapping Operators (MOps) [86], which have been developed throughout the TROPIC project,

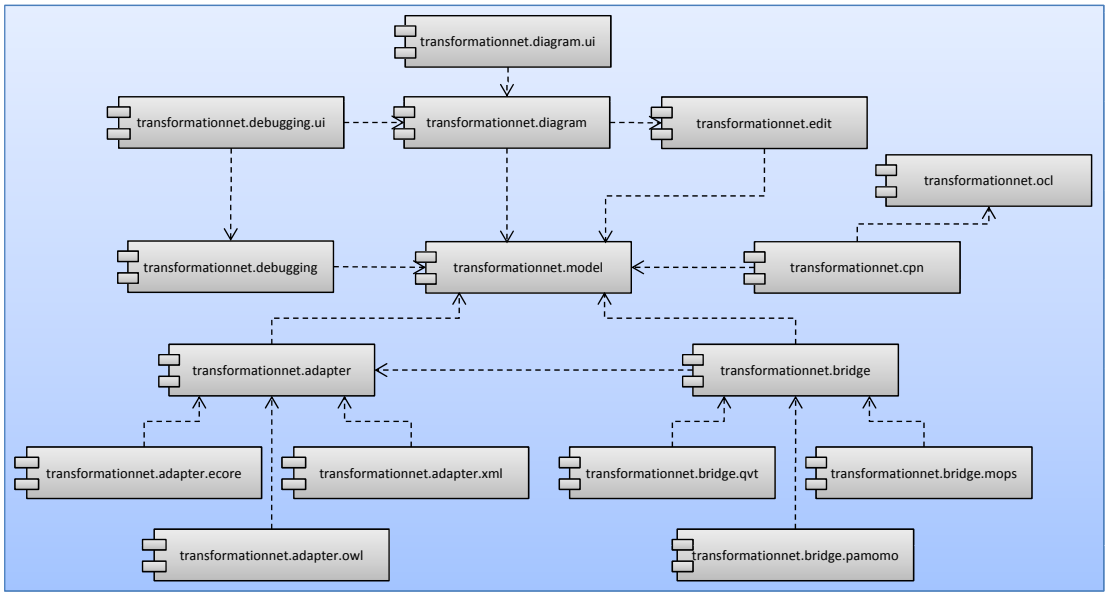


Figure 8.6: Components of the DEBUT prototype

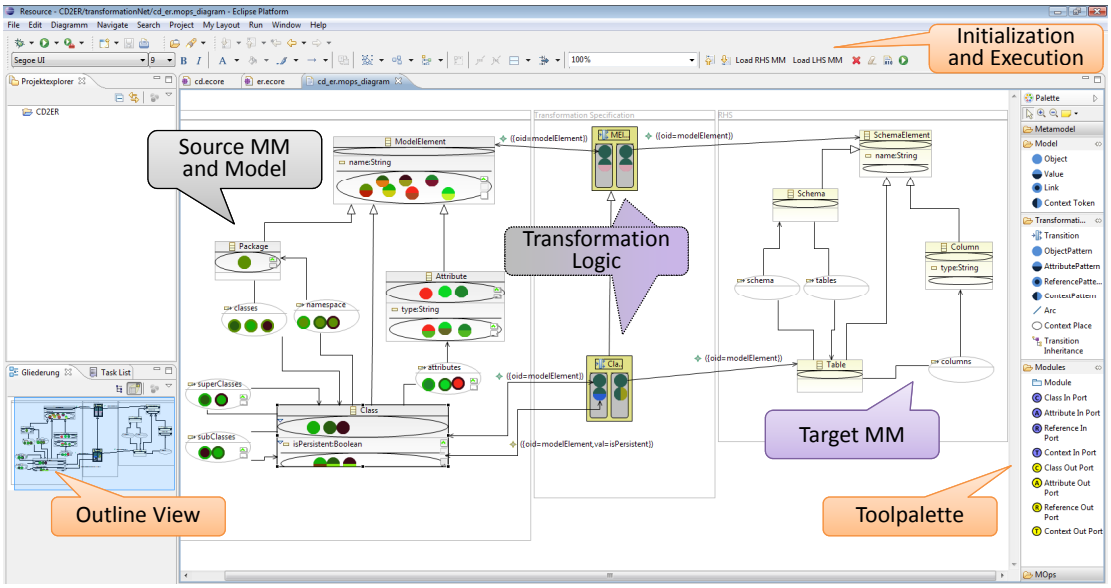


Figure 8.7: Screenshot of DEBUT

are provided. The main task of bridges is to provide a (bidirectional) transformation between the Transformation Net formalism and the according transformation language. For details on the provided bridges, especially concerning the included transformations, the reader is referred to Section 9.2.

Transformationnet.cpn. The compilation of Transformation Nets to CPNs is encapsulated within this plugin. It makes thereby use of the ASAP and Access/CPN framework [163, 164], which provides on the one hand a simulator component that is able to execute CPNs. It thereby makes use of the de-facto standard tool for creating, editing and simulating CPNs being CPN Tools. On the other hand, a PNML type for CPNs is provided by means of an EMF metamodel which is used as a target metamodel for the compilation of Transformation Nets into CPNs (as discussed in Chapter 6).

Transformationnet.ocl. In order to be able to evaluate OCL expressions within the CPN execution engine, the prototype makes use of the Comms/CPN library [48] provided by CPN Tools. This library allows for remote calls during execution of a CPN. Thus, the library enables to use a server which is able to evaluate the OCL expression and to send back the actual result of the evaluated expressions. The actual implementation of the server is implemented within this pluging. The server parses incoming requests and loads the according metamodels and models in order to evaluate the OCL expression.

8.2.2 Modes of Transformation Nets

The implemented prototype supports three different modes being (i) *raw*, to specify Transformation Nets, (ii) *transformation-based* to derive Transformation Nets from existing transformation specifications, and (iii) *contract-based* to derive a Transformation Net from the verifcation log

of PACO-Checker, whereby one mode has to be chosen when creating a new Transformation Net. The different modes are described in detail in the following.

Raw Mode

Transformation Nets not only provide a runtime model for model-to-model transformation languages, but they could also be used as a transformation language themselves, which is called *raw mode*. In this respect, it would be possible to specify a transformation logic by means of transitions. In a first step, the source and target metamodels as well as a source test model has to be provided. It is possible to create the source and target places representing the according metamodels as well as the tokens stemming from the source model by importing an existing metamodel or model (cf. *initialization buttons* in Fig. 8.7). Thereby, adapters are used which encapsulate the details on how to import a certain metamodel or model (cf. Fig. 8.8). As described in Section 4.3, not only Ecore based metamodels and models are supported but also those represented in XML or OWL. Further meta-metamodel formats can be supported by providing an according new adapter. In order to ease the specification of new adapters and the integration into the existing framework, according Eclipse extensions point are provided (named `SourceAdapter` and `TargetAdapter`). Furthermore, common base classes are provided which may be extended. Additionally, it is also possible to manually create the metamodel and the model by dragging according places and tokens from the *toolpalette* to the Transformation Net. These manually specified metamodels and models may then be exported to any of the supported formats. After having specified the metamodels and the source model, the transformation designer may start to create the actual transformation specification by means of a system of interacting transitions. Thereby, the transformation designer may make use of the provided debugging facilities already in the implementation phase.

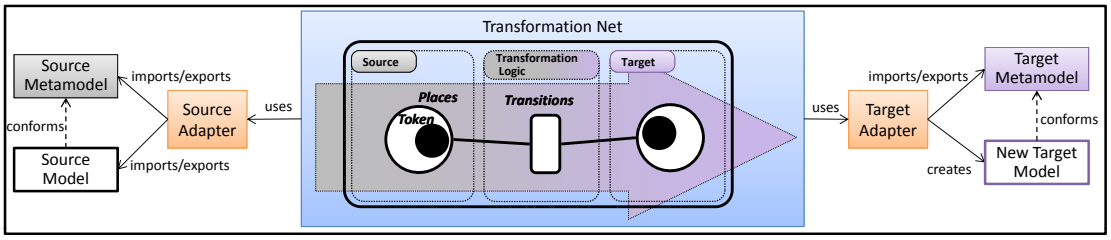


Figure 8.8: Transformation Nets Applied in Raw Mode

Transformation-Based Mode

Besides raw mode, the DEBUT prototype may also be used in the so-called *transformation-based mode*. This mode supports the debugging of declarative model-to-model transformation languages based on the runtime model. In order to achieve this, the transformation-based mode provides bridges to certain transformation languages (cf. Fig. 8.9). The main task of a bridge is to define a transformation from the actual transformation language to the Transformation Net formalism. In the current prototype, such transformations are implemented for QVT Relations

and for MOps. For details on the translation the reader is referred to Chapter 9. In case that changes in the Transformation Net should be propagated back to the transformation specification, e.g., when fixing a failure, an additional transformation is needed (or a single bidirectional one). The current prototype specifies separate transformations for every supported language, i.e., transformations from QVT Relations and MOps to Transformation Nets as well as the other way round are provided. Additionally, constraints may be defined in order to specify which elements of a transformation language may be edited, added and deleted in Transformation Nets. In this respect it is possible to constrain the possible changes in Transformation Nets, i.e., it may be forbidden to delete a module in Transformation Nets which represents a relation of QVT (cf. Section 9.2). Finally, a bridge encapsulates according adapters in order to be able to import and export the involved metamodels and models.

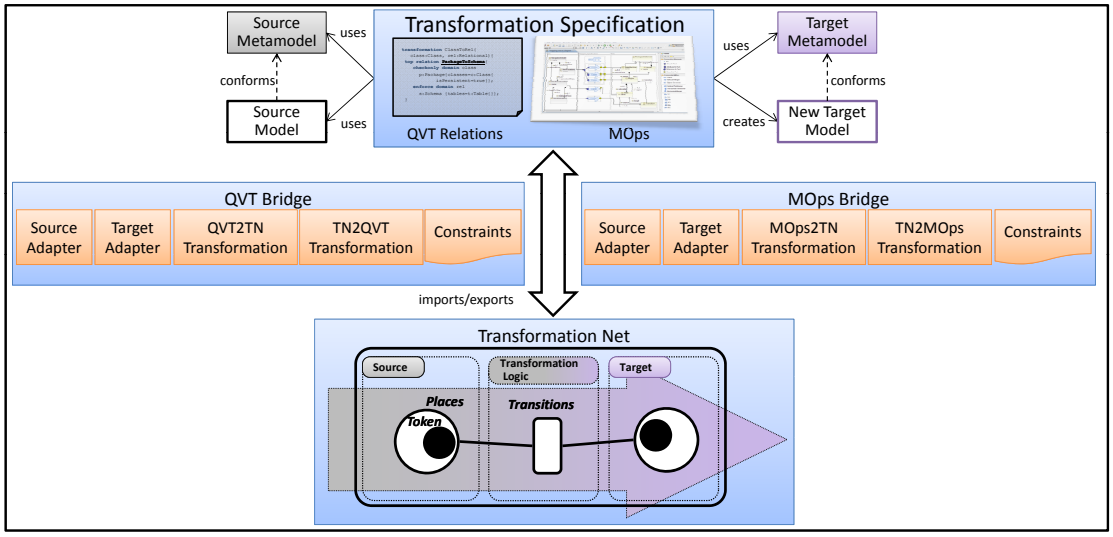


Figure 8.9: Transformation Nets Applied in Transformation-Based Mode

Contract-Based Mode

Last but not least, Transformation Nets provide the so-called *contract-based mode* which allows to combine PACO-Checker and DEBUT. PACO-Checker thereby produces a verification log as a result which can then be imported into DEBUT by an according bridge (cf. Fig. 8.10). Besides an error-trace, i.e., which source elements did not fulfill a certain contract, the verification log additionally contains references to the involved metamodels, the test input model as well as the transformation specification itself. The verification log is parsed by the `Verification Log Adapter` component. Equally to the transformation-based mode, in a first step the metamodels and the models are loaded and the transformation which translates the actual transformation specification to Transformation Nets is executed, e.g., a QVT Relations transformation specification is transformed to Transformation Nets. In a second step, the resulting Transformation Net is executed. Finally, the Transformation Net is initialized for re-enactment (cf. Section 7.3), i.e.,

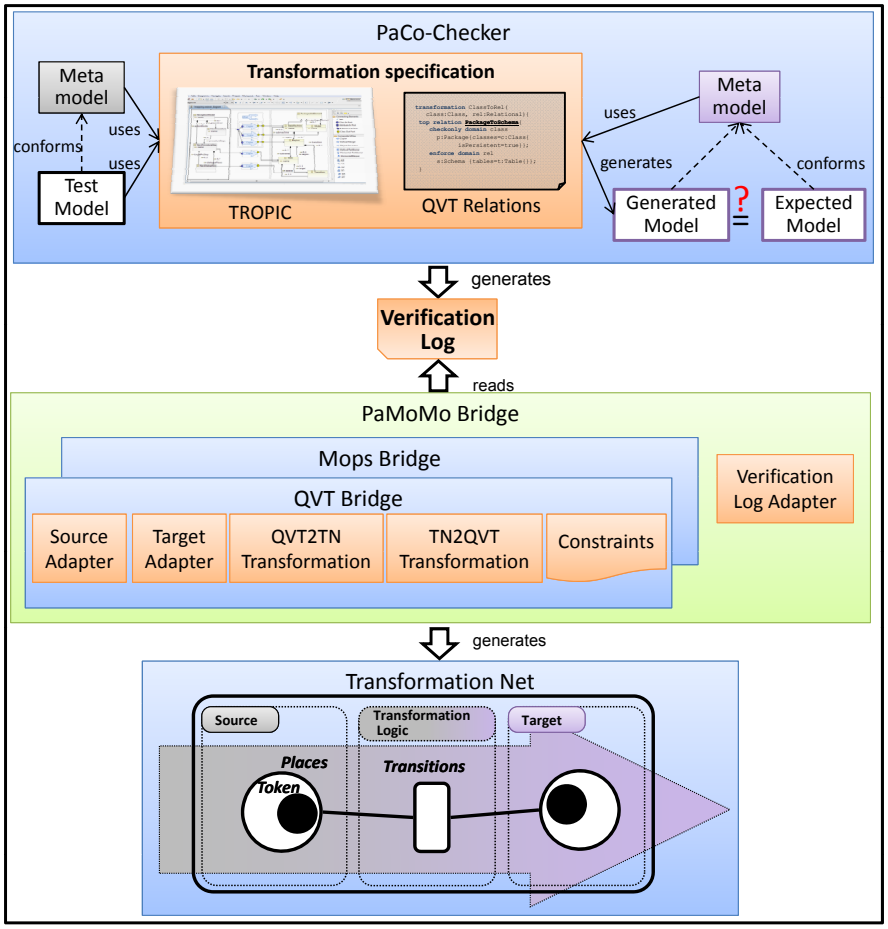


Figure 8.10: Transformation Nets Applied in Contract-Based Mode

the tokens which represent the model elements of the error trace are marked and the histories of the according transitions are reset in order to set a starting point for debugging.

8.2.3 Integration of CPN Tools into DEBUT

In order to be able to execute the specified Transformation Nets, they are compiled to CPNs as discussed in Chapter 6. Fig. 8.11 shows an overview on the technical realization thereof, mainly consisting of (i) the compilation of Transformation Nets to CPNs, (ii) the integration of the ASAP framework in order to execute the compiled CPN, and (iii) the evaluation of OCL constraints within CPN Tools.

Implementation of Compilation. As already mentioned, the ASAP framework provides a PNML type to represent the specifics of CPNs. For this, an EMF implementation is provided. In order to be able to represent the simulation results also in Transformation Nets, this metamodel has been extended with references to the according construct(s) in Transformation Nets, e.g., the CPN metamodel element `Transition` has been extended with a reference to a Transformation

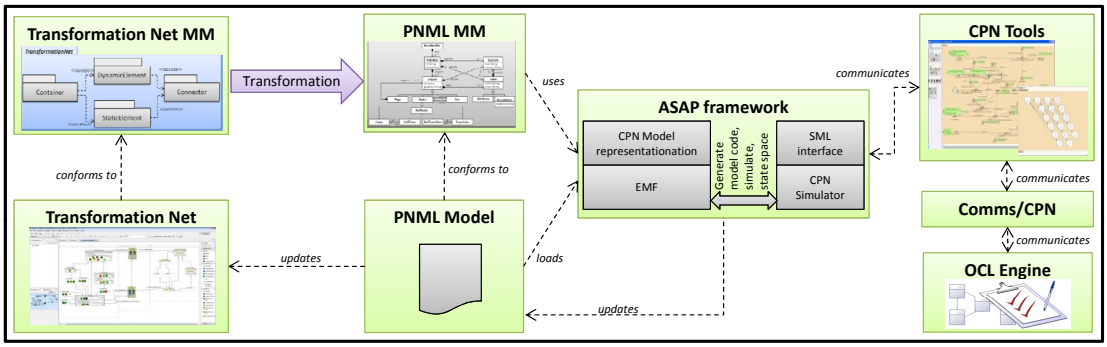


Figure 8.11: Integration of CPN Tools into DEBUT

Net `Transition`, which is the equivalent concept. In order to not "pollute" the original metamodel, subclasses of the to be extended concepts have been provided containing the according references. The compilation itself is written in Java due to the fact that, on the one hand complex calculations were needed, which would require imperative code anyway. On the other hand, an extensive and efficiently as well as easy accessible trace model is required, i.e., numerous methods are provided that provide flexible means to query the trace model. Finally, flattening of inheriting transitions would result in complex and hard to understand transformation code.

Fig. 8.12 shows the process of the compilation. In the initialization phase a `PetriNet` instance is created and it's root page is added. Furthermore, the color-sets as well as the required ML functions are defined. The actual compilation of Transformation Nets into CPN starts with the compilation of every `TNPlace` instance into a corresponding `Place` instance in CPN. These places are added to the root page of the Petri Net. After having compiled the places, the according objects, values and links are compiled to establish the initial markings of the places. Thereby, the tokens contained in places of a subtype are also considered in the marking of the supertype to allow for type substitutability, as already described in Section 6.2.

After having compiled the static parts of Transformation Nets, the dynamic parts are compiled. In a first step, for every module in a Transformation Net, according substitution transitions and subpages are created in the corresponding CPN. Afterwards, the inheritance hierarchy between transitions in Transformation Nets is flattened, following the principles presented in Section 6.4. The flattened `Transition` instances are compiled into CPN `Transition` instances and then added to the according page in the CPN. Finally, the patterns and arcs in Transformation Nets need to be transformed to according arcs and arc inscriptions. In case the pattern belongs to a transition which is contained in a module, additionally according ports are created and the assignments between ports and sockets are established (as discussed in Section 6.5).

ASAP Framework. The ASAP framework not only provides an Ecore based implementation of a PNML type for CPNs but additionally encapsulates components to simulate a CPN and

Figure 8.12: Compilation Process

to accomplish state space analysis (cf. CPN simulator and Standard Meta Language (SML) interface in Fig. 8.11). Basically, the framework provides an interface to CPN Tools and encapsulates the communication details, e.g., serialization of objects. In this respect an extensive framework is provided that allows to access the functionalities of CPN Tools from a Java program. After having compiled a Transformation Net to CPN the ASAP framework is used to check the CPN, i.e., it is analyzed if the resulting CPN is syntactically correct. If the net is correct, the simulator component may be used in order to achieve all enabled transitions. Since a CPN `Transition` instance contains a reference to its corresponding Transformation Net `Transition` instance, it is possible to highlight the enabled transitions also in the Transformation Net. Furthermore, the ASAP simulator provides means to either fire a single transition or to fire a certain number of transitions in a non-deterministic manner. If a transition fires, the markings of the according source and target places change. Therefore, the marking of these places is analyzed in order to accordingly update the tokens of the Transformation Nets. In this respect, the simulation of the Transformation Net is enabled.

Evaluation of OCL Constraints and Functions. In order to enable the evaluation of OCL expressions within CPNs, the prototype makes use of the Comms/CPN library [48]. This library allows for remote calls, which are used to evaluate the OCL expressions with the help of a remote server. In a first step the specified conditions and functions in Transformation Nets are specified in the guard in case of conditions or arc inscription in case of functions in CPN. Thereby it is assumed that OCL queries may be only specified on basis of the source metamodel, which is accessed in a read-only manner only. Since an OCL engine requires the according objects for the evaluation, in a first step currently bounded ids of the object are appended to the OCL expression. For example, the condition in Listing 8.1 requires the id of the bound `modelElement` as context (denoted by the `@modelElement` in the expression). Therefore, the according id bound to the variable `modelElement` is appended. Since several contexts are allowed, the according element is prefixed by its name followed by a colon. The `OCLEval` function sends the string via a socket connection to the server. If the server receives a message, it is first parsed, i.e., the actual condition or function is separated from the ids of the objects. The ids are then used to lookup the actual objects which are used to set the actual context objects for evaluating the OCL expression. The prototype makes then use of the Eclipse OCL framework to evaluate the expression. The evaluation result is returned to the CPN for further processing, e.g., enabling a transition if a condition evaluates to true.

Listing 8.1: Evaluation of OCL Condition

```
1 --bind a token to an OCL condition
2 OCLEval(''[@modelElement].isPersistent|modelElement:''^modelElement)]
```

8.2.4 Implementation of Debugging Features

After the general architecture of the DEBUT prototype has been described, this subsection gives an overview of the implementation of the debugging mechanisms presented in Chapter 7. In this respect, first the implementation of the mechanisms to detect code smells is presented followed by mechanisms for simulation-based debugging. Finally, the implementation of query-based and property-based debugging mechanisms is discussed.

Implementation of Mechanisms to Detect Code-Smells. In order to detect code-smells, OCL invariants are provided, which have already been presented in Section 7.1. If the Transformation Net is compiled to CPNs these invariants are automatically evaluated. The results are presented in the so-called `Code-Smells View` (cf. Fig. 8.13). In the example depicted in Fig. 8.13, the transformation designer can see that the source places `Package` and `Attribute` and the target places `Schema` and `Column` are not part of the transformation specification.

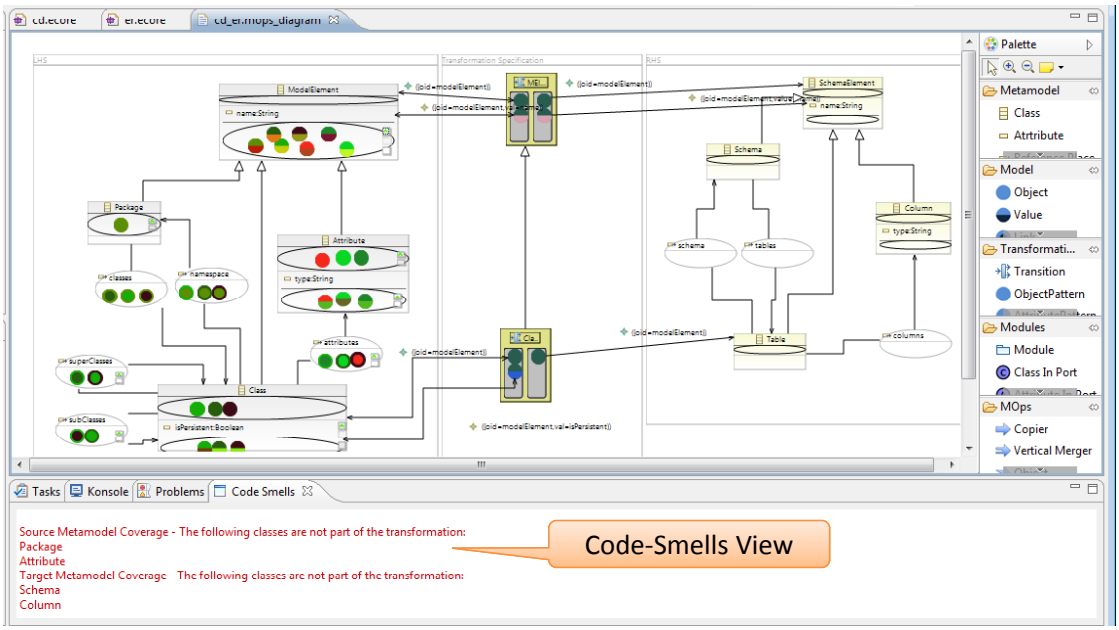


Figure 8.13: Screenshot of Mechanisms to Detect Code-Smells

Implementation of Simulation-Based Debugging Mechanisms. As described above, the ASAP framework allows to integrate CPN Tools into a Java application. To allow the transformation designer to compile the specified Transformation Net to a CPN, the editor's toolbar provides an according entry (cf. Fig. 8.14). The simulator component of the ASAP framework allows to execute a CPN, whereby the according results are visualized in the corresponding Transformation Net, i.e., the enabled transitions are highlighted in green, as can be seen in Fig. 8.14. During debugging the transformation designer may fire any of the currently enabled transitions by selecting the according context-menu entry. Furthermore, it is possible to select a possible binding in order to select specific model elements to transform, as depicted in Fig. 8.14. The editor's toolbox additionally allows the transformation designer to execute the transformation, i.e., the transformation is executed until it terminates or a breakpoint is reached. Finally, it is possible to clear all tokens and to load a new model or the revert the transformation, i.e., all produced tokens as well as the histories of the transitions are cleared and the transformation specification is set back to its initial state.

Implementation of Query-Based Debugging Mechanisms. Since the implementation of Transformation Nets follows the model driven approach, i.e., a transformation specification conforms to the Transformation Net metamodel, OCL can be used for debugging as explained in

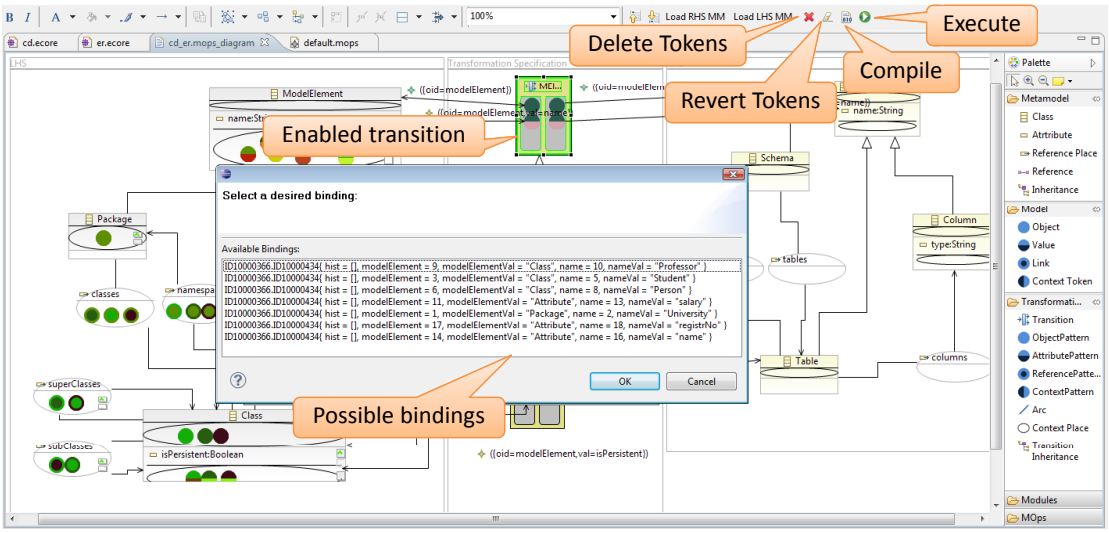


Figure 8.14: Screenshot of Simulation-Based Debugging Mechanisms

Section 7.3. In this respect, query-based debugging is enabled on the one hand by predefined OCL expressions, which are defined in terms of methods at the according metamodel element in order to reduce complexity and to provide the transformation designers with common debugging

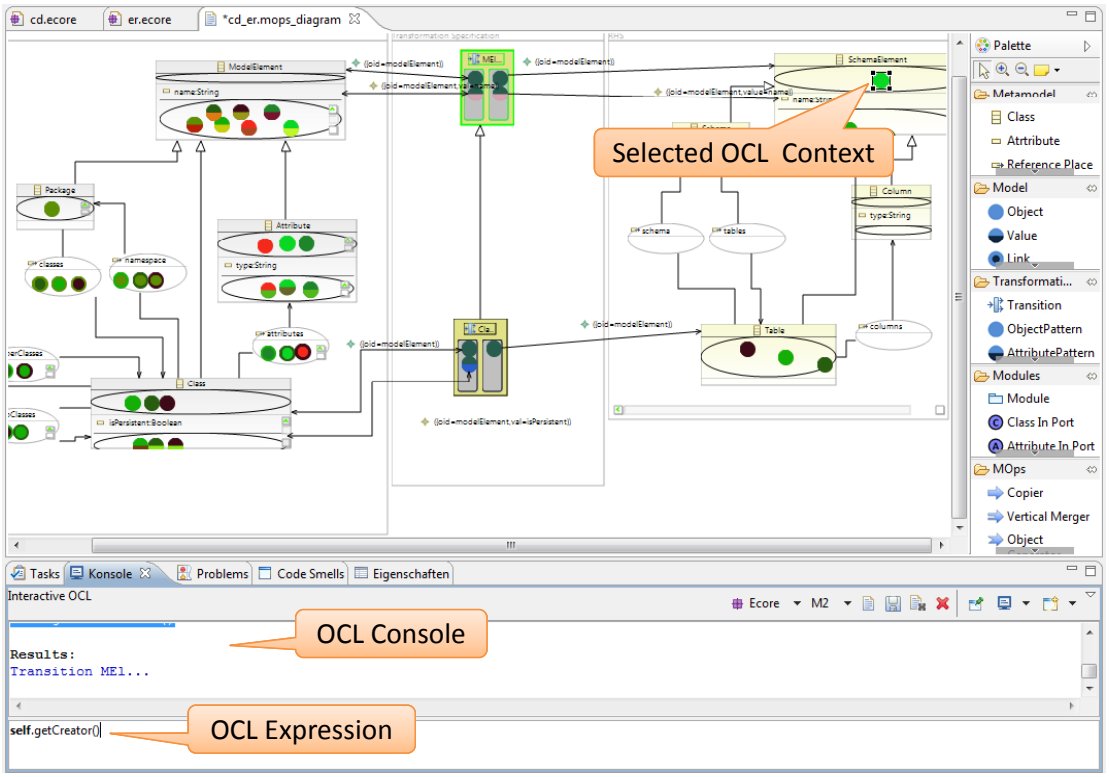


Figure 8.15: Screenshot of Query-Based Debugging Mechanisms

queries. On the other, these predefined methods may be combined with arbitrary OCL expressions in order to provide flexible means for backwards in time debugging and to not restrict the transformation designer to a limited amount of debugging questions. To specify the queries, the OCL console, which is provided by the Eclipse OCL framework, is used. In order to set the context of an OCL expression, arbitrary elements may be selected in the editor. In the example depicted in Fig. 8.15, the token contained in the place `SchemaElement` is selected. The OCL console depicted at the bottom of Fig. 8.15 is then used to query the `Transition` instance, which created the according token by calling the predefined method `getCreator()`.

Implementation of Property-Based Debugging Mechanisms. The ASAP framework not only allows to simulate a CPN but also provides possibilities to create the state space of a CPN in order to calculate behavioral properties, e.g., liveness properties or boundedness properties, as discussed in Section 7.4. For this, arbitrary ML statements may be executed. Consequently, statements are implemented that first calculate the state space of the underlying CPN. Furthermore, ML statements are implemented that query the state space. As a first step it is checked if the state space contains a dead state, i.e., if there exists a state without any outgoing arcs. If such a state is found, the according marking may be parsed and accordingly visualized in Transformation Nets. Furthermore, the DEBUT prototype allows to check for confluence of the specification, i.e., it is first checked if there exists a single home state and if this home state is equal to the dead state. As a further property it is possible to check for dead transition instances, i.e., if a transition has never been enabled. The results of the state space analysis are summarized in an according view as can be seen at the bottom of Fig. 8.16.

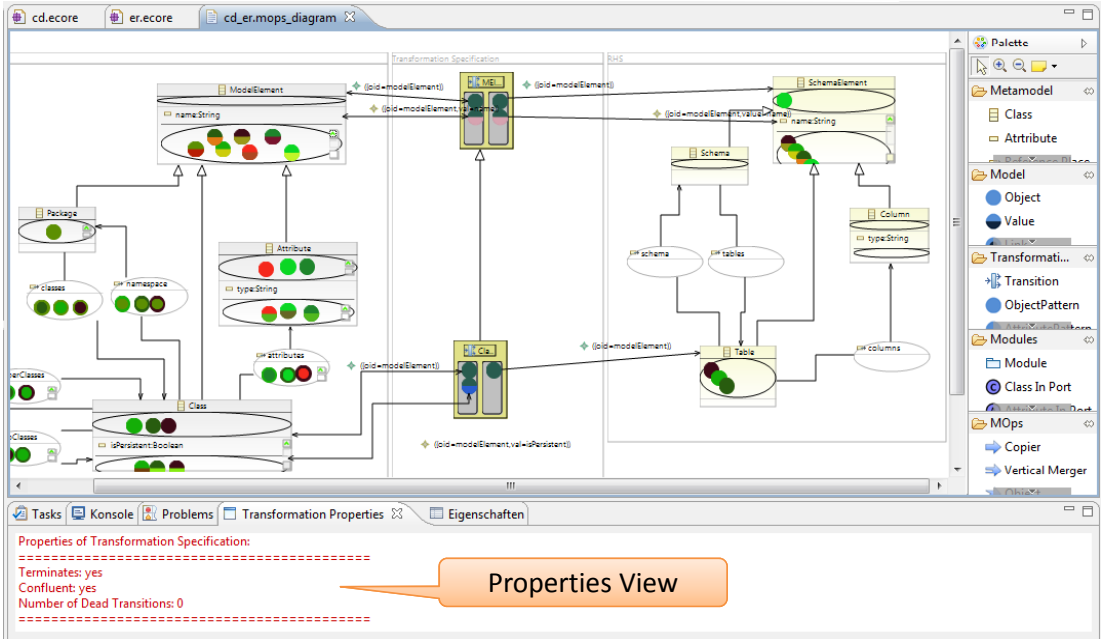


Figure 8.16: Screenshot of Property-Based Debugging Mechanisms

8.3 Summary

This chapter provided an overview of the implementation of the prototype. First, the implementation of the PACO-Checker tool was discussed which allows the specification of PAMOMO contracts, as well as their compilation and execution in QVT Relations in order to test a transformation specification against contracts. As a result, a verification log is provided which includes an error trace in case a certain contract is not fulfilled. Second, the DEBUT prototype was introduced, which provides an implementation of the Transformation Net runtime model and provides a graphical visual syntax in the form of an Eclipse editor in order to specify and edit Transformation Nets. Furthermore, the debugging features have been integrated into this editor. In order to benefit from the execution engine and the state space analysis methods of CPN Tools, the integration of the ASAP framework into the prototype was shortly discussed.

Chapter 9

Evaluation

If your experiment needs statistics,
then you ought to have done a better experiment.

— *Ernest Rutherford*

Contents

After having presented the concepts developed throughout the thesis, this chapter evaluates and critically reflects on them. The evaluation is structured along the three main contributions of the thesis. Thus, first, PAMOMO as a declarative language to specify visual transformation contracts is evaluated by means of case studies from different domains as well as a comparison to related work. Second, Transformation Nets as a runtime model for model transformations are evaluated. It is shown how transformations expressed using QVT Relations or the Mapping Operator (MOps) Language, a mapping language which was developed throughout the TROPIC project, can be represented by means of Transformation Net concepts in order to show the applicability and adequacy of the proposed runtime model. Finally, the debugging features are evaluated, again by means of a case study and a comparison to related work.

9.1 Evaluation of PaMoMo Contracts

In order to illustrate the usefulness of contracts, this section presents several case studies which were conducted in three application domains. The first one deals with the verification of the

transformation from PAMOMO into QVT Relations presented in this thesis. The second one is concerned with the verification of a complex transformation from a process-interaction simulation language [44] in the area of performance evaluation into CPNs [72]. Finally, the third one presents an application of contracts for third-party transformations, in particular the generation of visual editors from GMF models are tackled. These case studies show the versatility of the approach by the automated verification of an ATL transformation, a QVT Relations transformation, and the safe execution of a third party transformation (from which the source code is not available). In each case, the use of different features of PAMOMO are stressed.

9.1.1 Using PAMOMO to Verify its own Translation into QVT Relations

In this subsection, some patterns of the contract that helped in verifying the transformation from PAMOMO into QVT Relations are shown. The metamodels of both languages are depicted in Fig. 9.1. The PAMOMO has been already described in Subsection 3.2.5. A transformation (cf. class `RelationTransformation` specified QVT Relation makes use of several relations (cf. class `Relation`) which are defined between different metamodels (cf. class `TypedMode`). A `Relation` is connected to its metamodel by means of a `RelationDomain`, which contains so-called a `DomainPattern`. A `DomainPattern` is used to describe the actual elements that match for certain elements in the source model (checkonly domain) or produce certain elements in the target model (enforce domain). The configuration of the elements is described by means of the class `ObjectTemplateExp` and `PropertyTemplateItem`. With this example it is stressed that PAMOMO is independent from the language used to realize the transformations, since whereas in the running example a QVT Relations transformation is verified, here the translation was implemented with ATL.

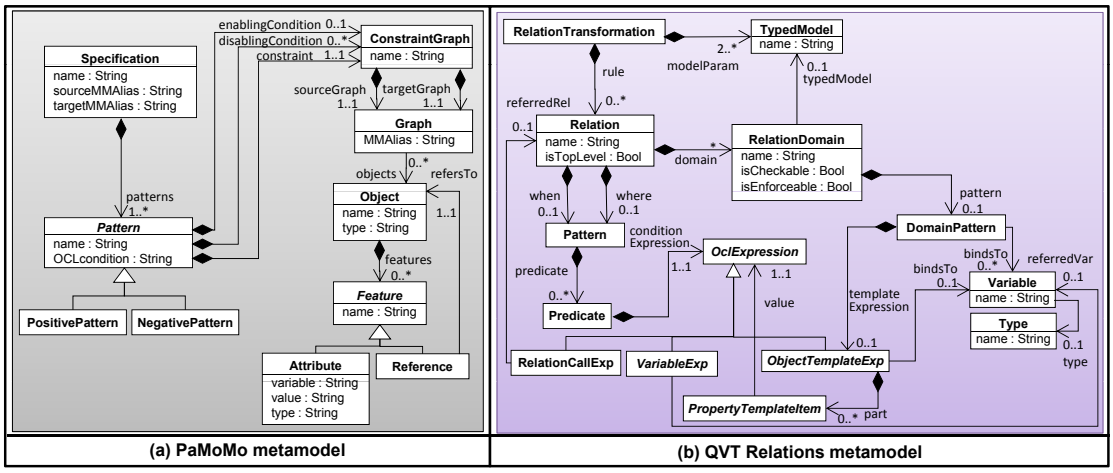


Figure 9.1: PAMOMO (left) and QVT-Relations (right) metamodels

The contract for the transformation contains invariants and postconditions, but it does not contain preconditions because the translation handles all features of PAMOMO. As an example, Fig. 9.2 shows an invariant addressing the translation of pre- and postconditions. These are

patterns with either the source or target graphs empty (i.e., the size of the set of objects either in the source graph or the target graph is 0, as checked by the condition). These patterns should be transformed into relations with two domains (since this is required by the used Moflon QVT Relations engine) but referring to the same `TypedModel` instance, as shown by the target graph object.

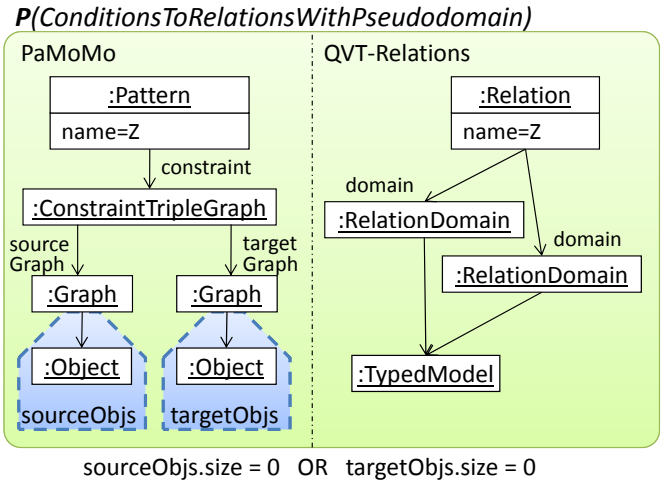


Figure 9.2: A Positive Invariant for PaMoMo-to-QVT-Relations

Fig. 9.3 shows another invariant stating that positive patterns of any type without enabling or disabling conditions (checked by the two disabling conditions) are transformed into a unique relation. Thus, the generated relation cannot invoke other relations in its *where* clause (i.e., the shown invariant) or *when* clause (checked by another similar invariant).

Finally, Fig. 9.4 shows two postconditions checking that all generated relation domains are

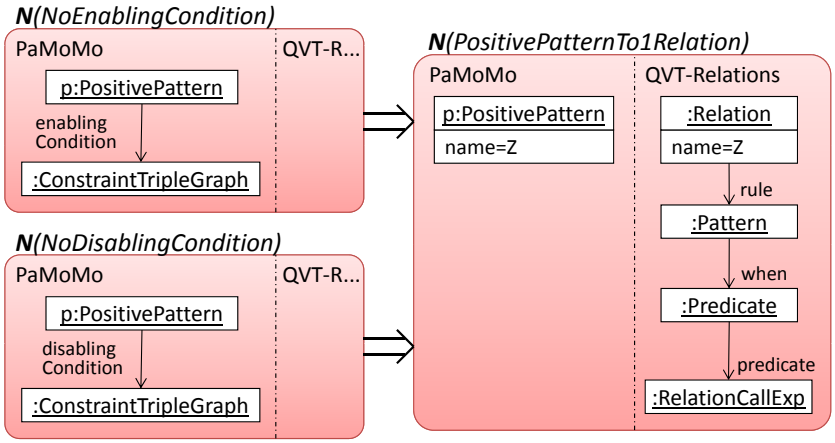


Figure 9.3: A Negative Invariant for PaMoMo-to-QVT-Relations

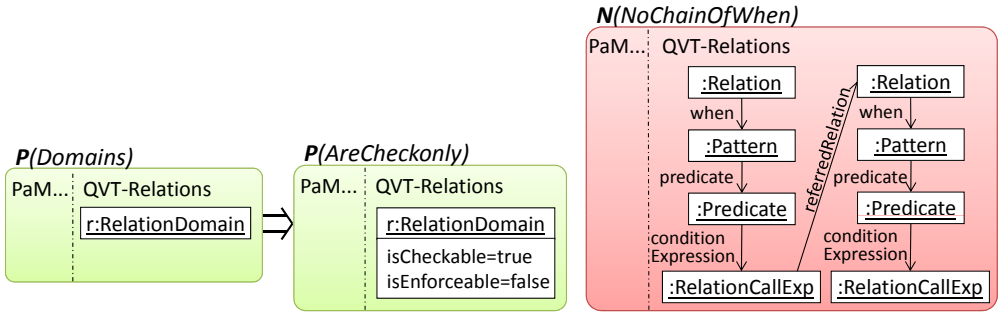


Figure 9.4: Two Postconditions for PAMOMO-to-QVT-Relations

`checkonly` (left), and that there are no chains of *when* relation invocations (right). Both are constraints of the models generated by the transformation.

9.1.2 From a Process-Interaction Language into Timed Coloured Petri Nets

The second case study deals with the translation from a process-interaction simulation language [44] in the area of performance evaluation into CPNs [72]. These two metamodels exhibited large heterogeneities and therefore it should be evaluated how invariants may cope with this situation. For modeling systems with the aim of simulating their performance, a language in the process-interaction simulation style [44] can be used. In this kind of languages, systems are modeled by processes made of interconnected blocks through which *transactions* flow.

Fig. 9.5 shows a process-interaction model. The two blocks to the left are *generators* of transactions. In particular, the upper left block produces a transaction of type 1 at each $[10, 20]$ time steps, with a transaction length having a uniform probability between $[120, 150]$. Similarly, the lower left block produces a transaction of type 2 at each $[12, 24]$ time steps, with a length having a uniform probability between $[140, 180]$. Both kinds of transactions arrive at the *advance* block (labeled "A"), which models a process with a delay given by a uniform probability in the interval $[2, 5]$. After this delay, transactions reach a *server* block with a parallelism of 3, meaning that the server can attend 3 transactions at the same time. Moreover, the server has a delay between $[4, 5]$. Then, a *type switch* block (labeled "type") selects the transactions depending on their type. Transactions of type 1 are routed onto a server with parallelism 2, while

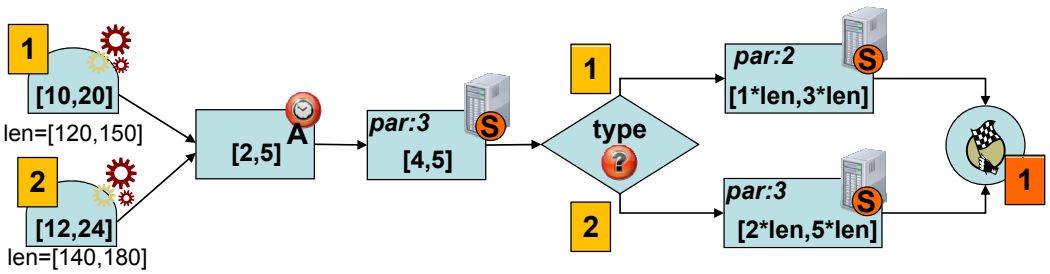


Figure 9.5: A Process-Interaction Model

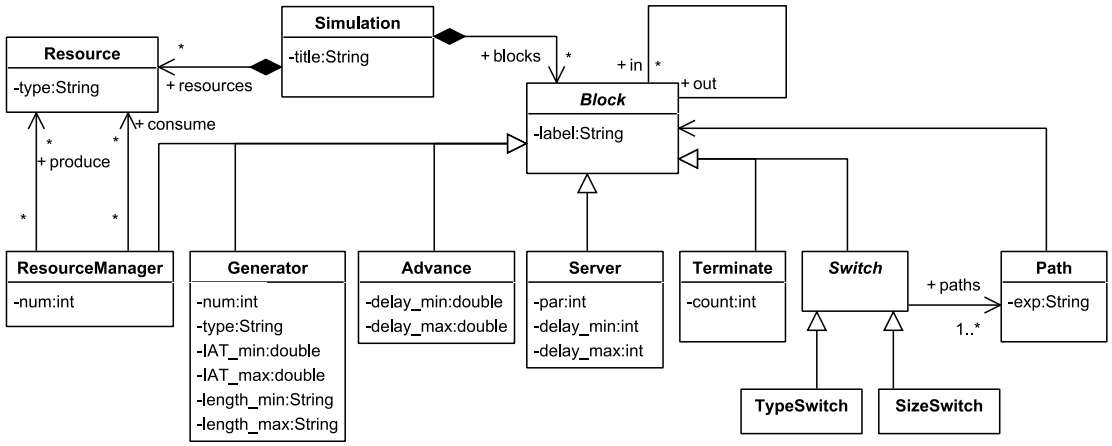


Figure 9.6: Metamodel of the Process-Interaction Language

transactions of type 2 are routed into a server with parallelism 3. Finally, transactions finish in a *terminate* block (which counts 1 each time a transaction arrives). Altogether, this model represents a client/server system that accepts two kinds of requests, processed on different servers.

Fig. 9.6 shows the metamodel for this process-interaction language. Thus, a `Simulation` model is made of `Blocks` and `Resources`. `Block` is an abstract class subclassed for each different kind of block.

In order to simulate and analyse process-interaction models, a transformation of these models into Coloured Petri Nets (CPNs) [72] has been built, which allows using tools like CPN Tools for this task. Please note that the example also makes use of the fact that CPN Tools supports time by attaching timestamps to tokens, which can be incremented by the transitions. PAMOMO has been used to express different requirements for this transformation. Fig. 9.7 shows some of the specified invariants. The one to the left expresses how parallel servers should be translated into CPNs. In particular, if the parallelism of the server is `P`, then we need to replicate `P` times the CPN structure inside the set of `servers` to the right. This is indicated by the expression `servers.size()=P`, i.e., the whole structure has to exist `P` times. The input and output blocks of the parallel server can be of any type, hence the invariant uses objects of type `Block` (represented by dotted rectangles) for them, to mean "any subclass of `Block`". Moreover, the labels `LS` and `LT` of these two blocks are used to locate the CPN places generated from them.

The upper right of Fig. 9.7 shows another invariant formalizing the translation of switches (both `TypeSwitch` and `SizeSwitch`). They should be transformed into places with as many output arcs as paths leaving from the switch. Finally, the bottom right of the figure shows an invariant describing the relation between the number of resources produced by a resource manager (with label `RM`) and the number of arcs that the corresponding transition should map to the place created for the resource. In particular, a correct transformation should produce as many arcs as the attribute `num` of the resource manager.

Altogether, in this complex case study the invariants made extensively use of sets. This is due to the fact that both metamodels exhibited large heterogeneities. In particular, it was often

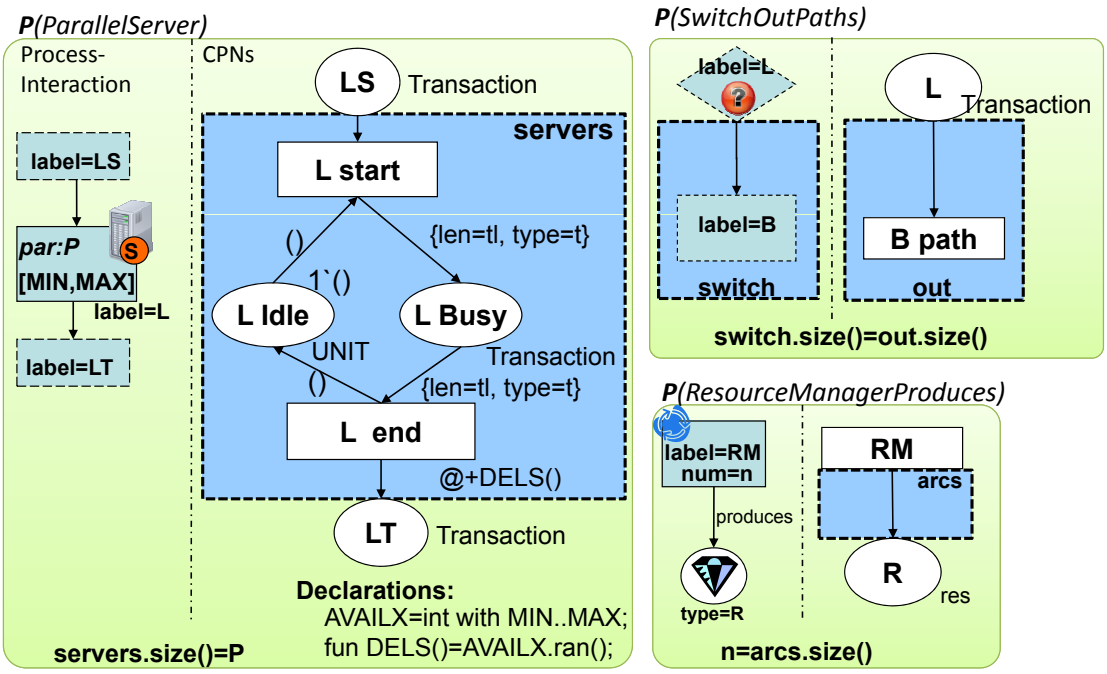


Figure 9.7: Invariants for: Translation of Parallel Servers (left), Translation of Switches (upper right), Translation of Number of Resources Produced by Resource Managers (bottom right)

the case that an attribute in the process-interaction language (like the parallelism in servers, or the resources produced by resource managers) had to be translated into a number of replicated structures in the CPN metamodel. Here, it can be benefited from the fact that patterns are declarative, so that complex structures can be easily described graphically, as opposed to textually encoding them using e.g., OCL navigation expressions.

9.1.3 Verification of Graphical Definitions in GMF

The Graphical Modeling Framework (GMF) [54] enables the "rapid" development of environments for visual languages. The approach taken is to specify different aspects of the editor using a set of interrelated models. The so-called `gmfgraph` model has a crucial role as it contains the specification of the graphical syntax of the language. However, only a tree-based editor is available for the specification of the figures of the concrete syntax, which is cumbersome and error prone. Therefore means are needed in order to verify if the specified model is correct. The `gmfgraph` model is then used (together with the other models) in a transformation to generate the so-called `gmfgen` model which is the basis for the final Java code generation of the editor.

A well-known problem is that if some model does not conform to a set of rules, the code generation produces erroneous code which may override a previous successful compilation. Although the framework validates some simple preconditions before the translation, like if all fields have meaningful values, no behavioral semantic contracts are checked. Therefore, designers of

GMF editors could greatly benefit from means to check whether their models conform to the set of GMF norms that ensure a successful compilation. In the following, contracts are specified for this purpose and some preconditions are discussed.

Layout constraints. The specifications of figures need to offer a certain `Layout`, e.g., a `GridLayout` providing a row/column oriented alignment. Typically, a complex figure consists not only of a single figure but also contains child figures, e.g., labels to visualize feature values. The actual visualization of the children figures can be constrained by means of `LayoutData`. However, the type of `Layout` for a figure, e.g., `GridLayout`, should correspond to the type of `LayoutData` for its children, e.g., `GridLayoutData`. To check this, the precondition in Fig. 9.8 may be used. The enabling condition selects figures with a certain `Layout` containing a child figure with some `LayoutData`. Then, the OCL condition in the precondition checks the compatibility of the `Layout` and the `LayoutData`.

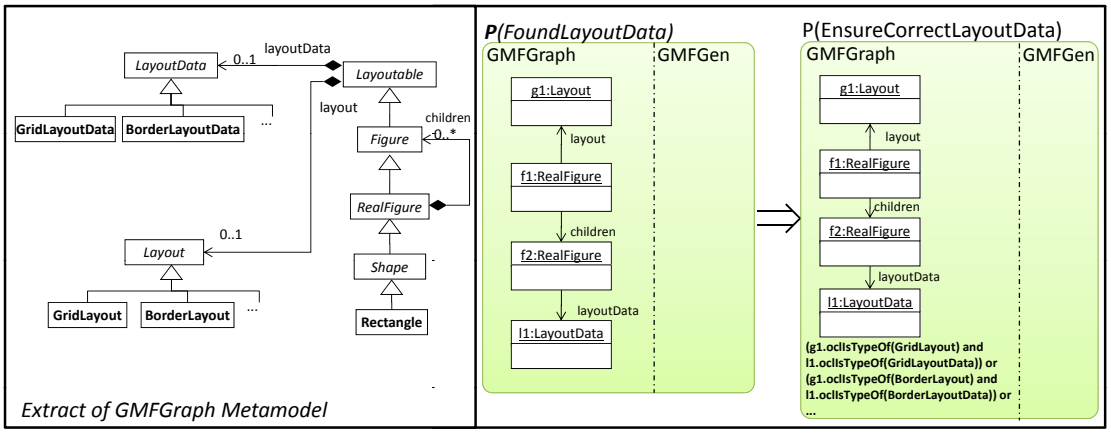


Figure 9.8: Precondition Checking Layout Constraints in GMF

Child access constraints. In order to be able to access the children figures of a figure in the `gmfmap` model, every `FigureDescriptor` needs to specify a `ChildAccess` for every one of its children. To be able to reuse a figure, e.g., if it is used several times in the concrete syntax, it is possible to use `Nodes` assigning a graphical representation to a certain metamodel element. Hence, if a `Node` refers not only to a `FigureDescriptor` but also to some `ChildAccess`, then the figure referred by the `ChildAccess` must be a child of the `FigureDescriptor`. In addition, the type of the `Node` has to correspond to the type of the child figure (e.g., in case of a `DiagramLabel`, the type of the child figure must be `Label`). These two conditions can be checked by using the precondition shown in Fig. 9.9. The second condition is encoded by the OCL expression in the pattern.

Hence, this example shows the use of PAMOMO to make explicit certain (non-documented) assumptions of transformations. Once these assumptions are encoded in the form of preconditions, they can be checked using PACO-Checker in order to avoid errors caused by the GMF compilation.

Finally, the kind of failures PaMoMo can detect is related to its expressiveness. There are some limitations concerning the specification of contextual conditions for a given property, as

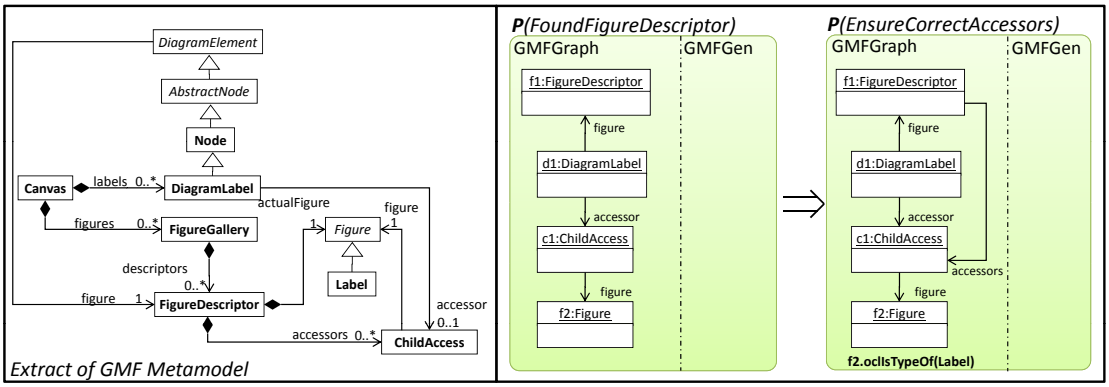


Figure 9.9: Precondition Checking Child Access Constraints in GMF

patterns only currently support conjunction of disabling conditions but not disjunction or any arbitrary boolean formula over disabling conditions, or nested (i.e., recursive) conditions. The expressiveness of the graphical part of the patterns is limited (less than first-order logic). For example, the absence of cycles of a given relation cannot be modeled. Nonetheless, in practice, we have found this expressiveness to be enough to build useful contracts declaring interesting properties for our transformations.

Regarding the scalability of the approach, it depends on the size of the tested input and output models as well as on the size of the patterns, as it is relied on a pattern matching mechanism. Thus, the smaller the models and patterns, the higher the performance. The size or complexity of the tested transformation implementation is not an issue though.

9.1.4 Comparison to Related Work

Even though the community has spent considerable research on verification and testing of transformations up to now, and although some approaches based on contracts have emerged recently, current approaches did not provide a high-level language which is able to express transformation properties in order to ease the specification of contracts. Therefore, in order to facilitate the specification of contracts, this thesis proposes a dedicated visual language to define transformation contracts which induces several advantages. First, the PAMOMO language is visual and enables a succinct expression of graph patterns, which otherwise would need to be encoded using navigation expressions in OCL, or complex expressions in the case of sets. In this respect, the pattern-based specification of PAMOMO contracts is intuitive to the transformation designer. Second, other languages like QVT Relations are less suitable than PAMOMO to express these contracts, in particular concerning the expression of negative information (a negative pattern produces two relations), or enabling conditions (which generates additional relations invoked from another relation). Third, PAMOMO's formal semantics enables also reasoning about metamodel coverage, redundancies, contradictions and pattern satisfaction. Fourth, the specification of the contracts is completely decoupled from the transformation implementation. This means that the contracts are independent from the specified transformation rules and from the trace model of a specific transformation execution. Finally, the translation of the contracts to QVT

Relations allows for dedicated feedback in terms of the model elements that did not satisfy a particular contract.

Compared to related work, a pure OCL-based approach, only provides true or false back as answer to the user, but no further information is accessible in standard OCL environments. Additionally, approaches based on OCL usually lead to complex constraints difficult to write in practice, and yielding verbose specifications [30], especially concerning invariants defining relations between input and output models. Furthermore, all mentioned approaches (except the model-fragment based ones [109,123]) define contracts based on the metamodel of the input and output models. In contrast to model-fragment based approaches, PAMOMO allows the definition of the contracts with a visual language but we refrain from defining the contracts for a particular test input model. Therefore, the presented approach is more general and allows to verify the specified transformation against arbitrary input models. Hence, this thesis contributes to fill this gap by providing a high-level, visual specification language and tool support to specify contracts. Since the specified contracts exhibit a formal semantics, reasoning at the level of patterns is enabled. Finally, contracts can be made operational by its compilation into QVT Relations.

Works dealing with testing object oriented systems with contracts report some limitations regarding the kind of failures contracts can detect [92]. For example, in object oriented systems method pre- and postconditions have difficulties in reasoning about the global state of an object or set of objects, as they are specified locally. For example, a prune method of a stack cannot have a trivial local contract checking if the removed element was previously inserted by a put method [92]. In contrast, transformation contracts are not specified at the rule level - as they are language-independent - and hence can be used to specify global transformation properties. In [50] it is also argued that detecting certain failures requires overly complex contracts, more than the method implementation itself. In PAMOMO, contracts can be made more precise by: (a) enriching a pattern with enabling or disabling conditions, (b) adding more objects in the source or target compartments of a pattern or (c) adding more patterns to the contract. For example, an enabling condition refines a pattern by making it fail on more models (hence, to potentially detect more transformation failures). However, it is up to future work to investigate the degree in which more complex contracts increase the effectiveness for failure detection (as in [22,92]).

9.1.5 Summary

In summary, in this subsection the appropriateness of PAMOMO contracts have been demonstrated by means of several case studies from different domains. In this respect it was shown, that the visual, declarative nature of PAMOMO eases the specification of contracts, e.g., in contrast to OCL or by (directly) using QVT Relations. After the evaluation of PAMOMO, in the following the proposed runtime model is evaluated.

9.2 Evaluation of Runtime Model

The second major contribution of this thesis is the provision of a runtime model for declarative, rule-based model-to-model transformation languages. Chapter 4 introduced the con-

cepts of Transformation Nets in general and Chapter 5 focused on rule inheritance in particular. In order to evaluate the presented concepts, in the following it is first shown, how the operational semantics of QVT Relations may be represented in Transformation Nets. Second, it is shown how the operational semantics of the declarative mapping language called Mapping Operators (MOps), may be specified by means of Transformation Nets. The representation of the operational semantics of both, QVT Relations and MOps, allows the reuse of the debugging features provided by the Transformation Net formalism, which will be evaluated afterwards.

9.2.1 Translating QVT Relations to Transformation Nets

In order to show the applicability of the presented runtime model, this section evaluates if the provided concepts are able to represent the operational semantics of current model-to-model transformation languages. As a concrete example, the operational semantics of QVT Relations, which has been proposed as *the* standard declarative model transformation language by the OMG, should be represented by the concepts provided in Transformation Nets. This allows to reuse the debugging features of Transformation Nets also for QVT Relations.

Since transformations written in the QVT Relations language consist of declarative relations between metamodels, *unidirectional* as well as *bidirectional* transformations are supported, although the actual execution requires to specify a direction. Moreover, QVT Relations supports *check* and *enforce* semantics, differing in if required changes on the target side are just reported or actually undertaken, thereby supporting incremental updates which can theoretically be specified on rule level. The semantics of check and enforce, especially in combination with bidirectional model transformations is not clearly defined as stated in [144]. Furthermore, the QVT standard defines the operational semantics of QVT Relations twofold, firstly in natural language and secondly by a translation to QVT Core, being incompatible to each other [145]. This situation led to different implementations of the operational semantics in different tools, e.g. concerning the realization of *when* and *where* clauses. To circumvent these deficits of the QVT standard, the example as well as the translation presented in the following are based on the operational semantics of the mediniQVT[1] implementation. Furthermore, the focus is on an exogenous batch transformation scenario which creates a new target model out of an existing source model. In case of bidirectional specifications, two different Transformation Nets are derived, one for every execution direction. Thereby, it is assumed that all relations specify a *checkonly* semantics for source model elements and *enforce* semantics for target model elements.

9.2.1.1 Translating QVT Relations to Transformation Nets

In order to present the translation of QVT Relations to Transformation Nets, the running example is used. Fig. 9.10(a) shows an extract of the QVT Relations code whereas Fig. 9.10(b) shows the according representation in Transformation Nets. In the following, the translation is described in detail.

Representation of Source and Target Metamodels and Models. QVT Relations as well as Transformation Nets provide containers (cf. class `RelationTransformation` and `Net`,

[1] http://projects.ikv.de/qvt

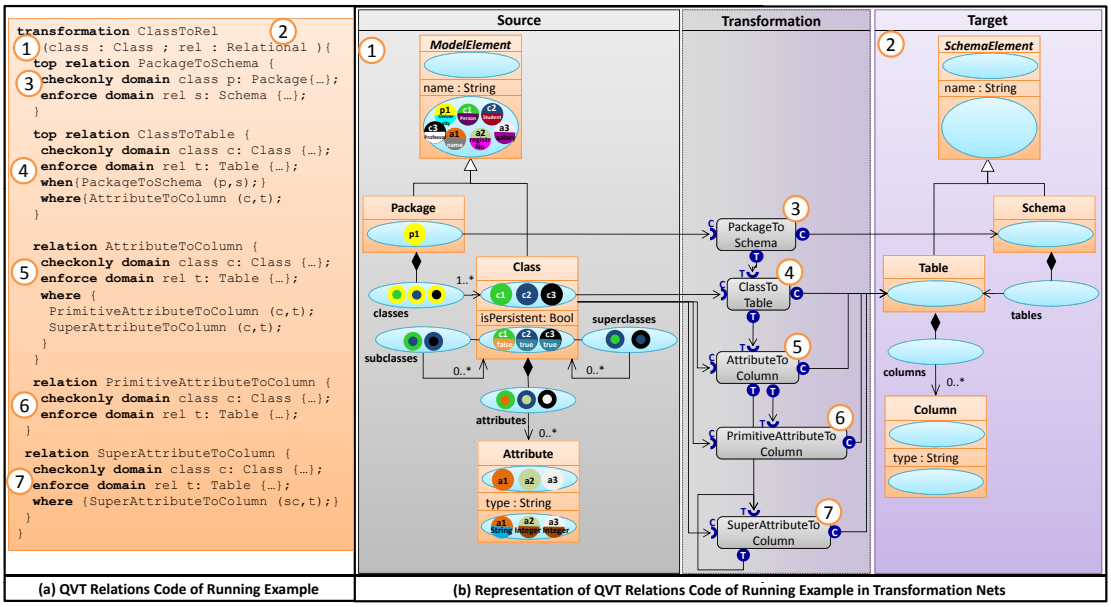


Figure 9.10: Representation of QVT Relations Code in Transformations Nets (Blackbox-View)

respectively in Fig. 9.11) for aggregating metamodels, models and transformation logic. Therefore it is natural to represent instances of `RelationTransformation`, i.e., transformations, by instances of `Net` and `TransformationSpecification` in Transformation Nets. Every QVT Relations transformation specification requires two so-called candidate models, which are given as parameter to the transformation specification and are represented by instances of `LHS` and `RHS` in Transformation Nets, providing containers for places representing source and target metamodel (cf. reference `RelationTransformation.modelParameter` and ① and ② in Fig. 9.11). These parameters, e.g., `class` and `rel` represent references to the according source and target models and their respective metamodels (cf. metaclass *TypedModel*). These references have to be set by the transformation designer before executing the transformation specification, i.e., in the runtime configuration of mediniQVT. These information is used to derive according instances of `TNPlaces` of the metamodels and `Tokens` from the models, following the principles described in Section 4.3.

Representation of Transformation Logic. QVT Relations uses `Relations` in order to specify the transformation logics. Thereby, `Relations` act as a container to encapsulate so called `DomainPatterns` which match for a source element and create a target element. To incorporate the involved metamodels, QVT Relations uses `RelationDomains` which bind `Relations` to the source or target metamodel. Consequently, `Relation` instances are represented as a `Module` instance with a contained `Transition` instance in Transformation Nets. The root `DomainPatterns`, which have to target metamodel classes, are represented as explicit `ClassPorts` in the blackbox view and as according arcs originating from the corresponding metamodel element, as can be seen in Fig. 9.10 and Fig. 9.11. In this respect, `Arcs` in Transformation Nets are used to represent `RelationDomains`.

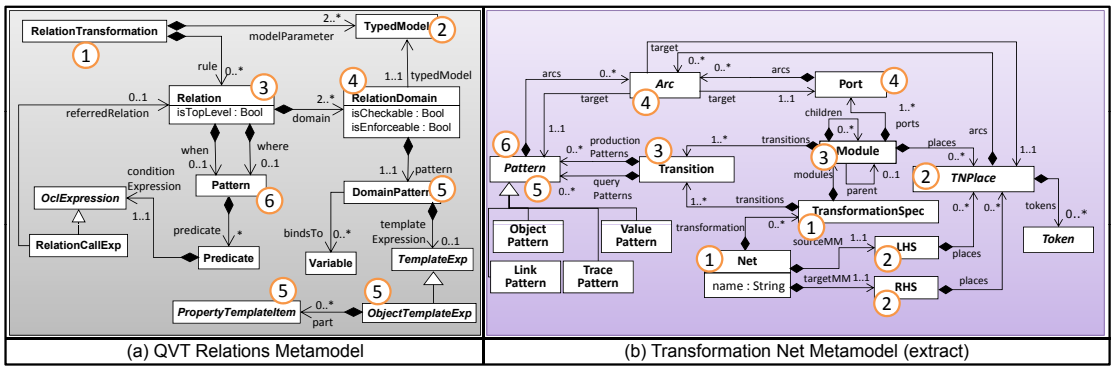


Figure 9.11: Correspondences between QVT Relations and Transformation Nets

To specify the actual elements that should be matched, i.e., preconditions of a transformation rule, and which elements should be created, i.e., postconditions of a transformation rule, so-called `DomainPatterns` are used in QVT Relations. `DomainPatterns` must build digraphs conforming to the used metamodel (cf. Fig. 9.12 depicting the `DomainPatterns` of the relation `PackageToSchema`), expressing correspondences between source and target metamodel elements. Unfortunately, this correspondence is hard to grasp. To get a visual clue which source element is transformed to which target element, Transformation Nets represent the nodes of such a digraph graphically (cf. Fig. 9.12) by means of `Query-` and `ProductionPatterns` whereby every node is connected to a certain source or target place. Following the principles of Transformation Nets presented in Chapter 4, correspondences between source and target element are expressed by equally colored patterns.

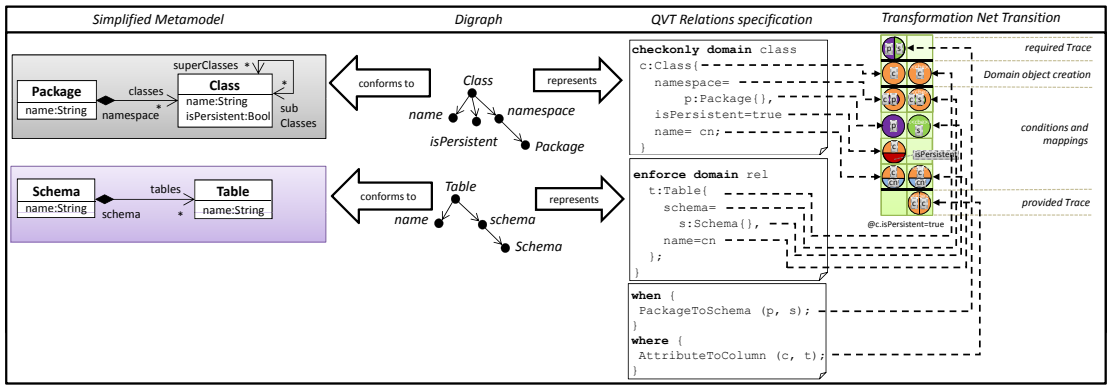


Figure 9.12: Dependencies between Metamodels, QVT, and Transformation Nets

In QVT, `DomainPatterns` specify the selection of model elements forming, as mentioned before, a digraph conforming to a metamodel using the specified domain object as root node (cf. Fig. 9.12). The graph consists of objects (cf. class `ObjectTemplExp`), attributes and links—both represented by the class `PropertyTemplateItem`. Starting from the do-

main object (i.e., the digraph's root node of Fig. 9.12) which is represented in Transformation Nets by an `ObjectPattern` within transitions, navigation in the graph is enabled using `LinkPatterns`, e.g., the `namespace` link in Fig. 9.12. Primitive values in case of attributes are represented by according `ValuePatterns`, e.g., the `name` attribute in Fig. 9.12. Thus, instances of `ObjectTemplExp` in QVT Relations are expressed by `ObjectPatterns` and instances of `PropertyTemplateItem` are expressed either by `LinkPatterns` or by `ValuePatterns` in Transformation Nets, depending if the `PropertyTemplate` in QVT Relations either refers to a reference or an attribute (cf. Fig. 9.13). The variables in the QVT Relations specification are then used as a variable in the according pattern.

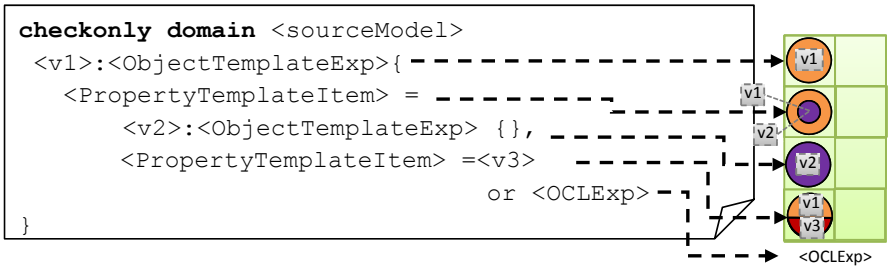


Figure 9.13: Schema of Translation.

QVT Relations allows to specify *DomainPatterns* containing references of the target model having a multiplicity greater than one, e.g., a `Schema` can contain an arbitrary number of `Tables`. To ensure that the target domain object, e.g., a `Schema`, is created only once, a QVT Relations transformation engine examines the trace information and checks if the according target element has been created before, i.e., it makes use of a *check before enforce semantics*. The check before enforce semantics ensures that, if an object matching the constraints in a relation already exists in the to be generated target model, this object will not be newly created. QVT Relations furthermore allows defining equality of objects by means of *keys*. In order to represent this behavior accordingly in Transformation Nets, every production pattern is marked with the check before enforce flag if no specific key is defined. In case a key is defined for this object, those patterns that represent elements of the key are added int the `Transition.key` array, e.g., if a key specifies, that `Table` instances should be differently named, then the according `ValuePattern` producing the values and its corresponding `ObjectPattern` would be marked as check before enforce and added to the `Transition.key` array.

Representation of When and Where Clauses. When and where clauses in QVT Relations fulfill two tasks, (i) they allow to pass elements between relations and (ii) they may be used to specify further constraints by means of OCL. In order to allow for interconnections between `Modules` and their contained `Transitions` in Transformations Nets (which represent Relations of QVT Relations in Transformation Nets), `TracePatterns` and `TracePorts` are used in Transformation Nets. To be more precise, in case of a when clause, the `Transition` instance contains an according `TracePattern` that queries the according trace place of the relation which is called in the when clause, as can bee seen in Fig. 9.12 and Fig. 9.14. Where clauses are represented by production `TracePatterns` and according `TracePlaces`. These `Trace-`

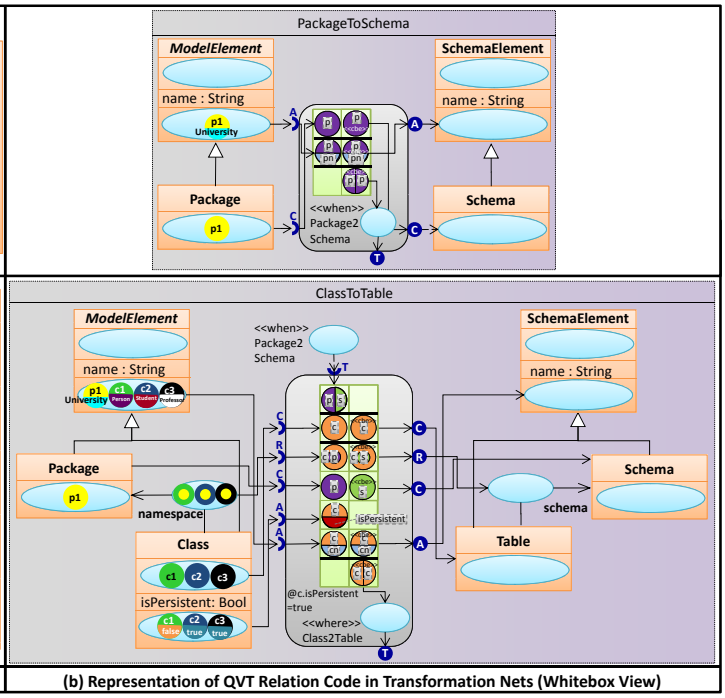
Places are then queried by dependent transitions. In case the when and where conditions contain further constraints, they are transformed into according OCL conditions attached to the according transition.

9.2.1.2 QVT Relations to Transformation Nets by means of the Running Example

Fig. 9.14 shows the translation of the QVT Relations specification using an extract of the running example (cf. Fig. 9.14(a)) to Transformation Nets in its whitebox view (cf. Fig. 9.14(b)). The specification of the transformation in QVT Relations consists of two relations for establishing the one-to-one correspondences. The PackageToSchema relation matches for packages and their names and produces equivalent schemata and names thereof. The relation ClassToTable matches for persistent classes contained in a package as well as their names and creates a table labeled with the class name. The reference to the according schema is set by calling the PackageToSchema relation in the when clause of the ClassToTable relation. Even this simple example raises questions concerning specification and execution of the transformation, e.g., what happens if there are no persistent classes in a schema or in which order are the relations actually executed and the model elements created since both relations are marked as top? In order to clarify these and further debugging questions, the debugging features of Transformation Nets should be available for QVT Relations as well, i.e., the QVT Relations specification needs to be translated to Transformation Nets as depicted in Fig. 9.14(b).

```
4  // map each package to a schema
5  top relation PackageToSchema {
6   pn: String ;
7   checkonly domain class p: Package {
8    name =pn
9   };
10   enforce domain rel s: Schema {
11   name =pn
12  };
13 }
```

```
15 // map persistent class to a table
16 top relation ClassToTable {
17  cn: String ;
18  checkonly domain class c: Class {
19   namespace =p: Package {},
20   isPersistent =true ,
21   name =cn
22  };
23  enforce domain rel t: Table {
24   schema =s: Schema {},
25   name =cn
26  };
27  when {
28   PackageToSchema (p, s);
29  }
30  where {
31   AttributeToColumn (c, t);
32  }
33 }
```



(a) QVT Relations Code of Running Example

(b) Representation of QVT Relation Code in Transformation Nets (Whitebox View)

Figure 9.14: QVT Code and Corresponding Transformation Net (Extract)

The translation follows the principles presented above. In a first step the domain objects, i.e., the roots of the digraphs, are mapped. Therefore, the upper `Transition` in Fig. 9.14(b) matches for `Package` instances whose `Package.name` attribute is not null. In order to keep the gap between the QVT Relations specification and the resulting Transformation Net low, the variables of the patterns are equally named as the variables in QVT Relations. Furthermore, the order of patterns is equal to the order of definitions in QVT Relations. Equally named variables that are used in the `checkonly` and `enforce` domain denote assignments. Therefore, the according patterns are equally colored in the Transformation Net, e.g., the `ValuePatterns` representing the `name` attribute. This is different for objects, which need not to be assigned to a common variable, e.g., although `Package` instances should be translated to `Schema` instances, two different variables `p` and `s` are used in the QVT Relations specification. In order to better visualize this correspondence and to provide visual trace information, in Transformation Nets the production `ObjectPattern` is nevertheless equally colored as the according query `ObjectPattern`, e.g., the `ObjectPatterns` are both labeled with the variable `p` in the upper `Transition` in Fig. 9.14(b). Finally, these trace information is also made explicit by means of a production `TracePattern` and an according `TracePlace`.

Concerning the translation of the relation `Class2Table` the same principle is followed, i.e., according patterns are created for the variables used in the QVT Relations specification. Please note that the `ObjectPatterns` querying for instances of `Packages` and producing instances of `Schema` offer different colors and variables (`p` and `s`) since these variables are also used in the `when` clause of the relation, i.e., it is required that a certain `Schema` instance has already been created for a certain `Package` instance (since a when clause represents a precondition). Therefore, the transition uses a query `TracePattern` which queries the trace information of the relation called in the when clause, e.g., the trace information of the transition representing the relation `Package2Schema` in the example depicted in Fig. 9.14(b). Furthermore, please note that in this case it is important, that the production `ObjectPattern` producing `Schema` objects is marked as check before enforce. If this is the case, existing objects are reused instead of being created again, i.e., an existing `Schema` instance is not newly created but only the `Table.schema` links in the according `Table` instance are created, since no link with equal source and target object exists already.

In summary, this section showed that QVT Relations specifications may be represented by means of Transformation Net concepts. Nevertheless, currently queries in QVT Relations as well as inheritance between transformations, which is allowed in QVT Relations, and sequences and set of patterns are not handled in the presented transformation. In this respect, the debugging facilities of Transformation Nets are also available for QVT Relations, although there is a gap between the representation in terms of Transformations Nets and the representation of the actual QVT Relations code. Nevertheless, the similar operational semantics is achieved, which is made explicit to the transformation designer. An evaluation thereof will be presented in Section 9.3.

9.2.2 Translation of Graph Transformation Languages to Transformation Nets

Several related work already exists that relates the domains of Graph Grammars to Petri Nets [7, 103]. It has already been shown that the concepts of one paradigm may be expressed in the other paradigm. Since Transformation Nets base on CPNs it is also possible to represent Graph

Grammars in Transformation Nets. As a proof of concept, the implementation of AGG [147] has been mapped to Transformation Nets on a conceptual level by means of first case studies.

Concerning the translation of the type graph (which represents a unified metamodel, i.e., source metamodel, target metamodel and a custom trace model), certain naming restrictions have to be followed. This means that elements of the source metamodel are assumed to offer the appendix `_source`, those of the target metamodel `_target` and those of the trace model `_trace` to derive the according types of places, as can be seen in Fig. 9.15(a). The LHS part of the graph transformation rule may be represented by according `QueryPatterns` in Transformation Nets whereas the RHS part of a graph transformation rule may be represented by according `ProductionPatterns`. Please note that, in contrast to the translation of QVT Relations, the visual trace is lost for objects which can be seen in the Transformation Net as the query `ObjectPattern` offers the variable `p` whereas the production `ObjectPattern` offers the new variable `s` (cf. Fig. 9.15(b)) . This is since newly generated elements in graph transformations have to offer a new id in the RHS. The assignment of values, e.g., the assignment of the value of the attribute `Package.name` to `Schema.name` can be achieved by equally colored tokens (cf. Fig. 9.15). Negative application conditions (NACs) may be encoded by means of negative patterns which are translated to a pattern that realizes inhibitor arcs in CPNs (cf. Section 6.3) and thus follow the principles suggested in [7, 103]. Nevertheless, NACs that solely concern the trace model, e.g., in the example depicted in Fig. 9.15(a) the NAC solely ensures that `Package` instances that have already been translated to `Schema` instances are not

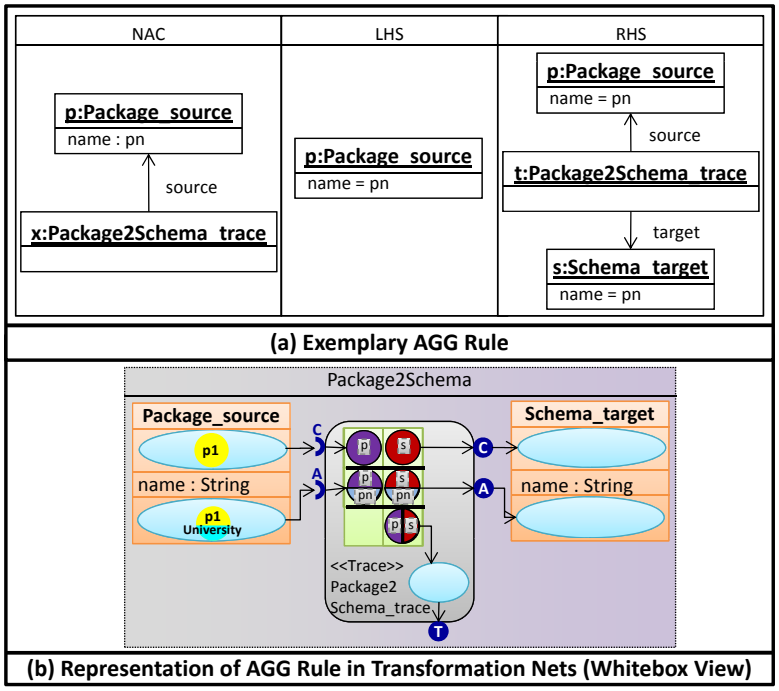


Figure 9.15: AGG Code and Corresponding Transformation Net

matched again, are treated differently. This behavior is automatically ensured by Transformation Nets by the history concept and therefore such NACs need not to be represented explicitly. A challenge that arises is the representation of conditions and functions since AGG uses Java for this purpose. Also the current prototype only supports OCL as an inscription language, on a conceptual level also Java may be supported as inscription language in Transformation Nets as well, i.e., instead of a server that is capable to evaluate OCL expressions, a server that evaluates Java expressions would be needed which is then called from the compiled CPN by using the Comms/CPN library. Concerning the interplay between the specified graph transformation rules, AGG allows to specify *layers* to order the application of rules. The representation of this concept (as well as general concepts concerning the actual execution semantics of graph transformations) is still an open issue, as discussed in Chapter 10.

In summary, based on existing literature, first case studies for translating graph transformations specified in AGG to Transformation Nets have been conducted, which provided promising results. Nevertheless, in order to fully evaluate the applicability a prototype is required, which is considered as a point of future work (cf. Chapter 10).

9.2.3 Translating Mapping Operators to Transformation Nets

In the last years, approaches arose that propose to specify model transformations by means of abstract *mappings* being a declarative description of the transformation, as known from the area of data engineering [14]. A concrete implementation of such a mapping approach has been presented in [86], where so called Mapping Operators (MOps) have been proposed, being the second major outcome of the TROPIC project. The operational semantics of MOps is defined by means of Transformation Nets, i.e., Transformation Nets act as a semantic domain for the mapping language. In the following, first, a short overview on MOps is given before showing the specification of the operational semantics by means of Transformation Nets.

9.2.3.1 Mapping Operators in a Nutshell

The main idea of mappings is to abstract a model transformation problem from a concrete transformation language, allowing the transformation designer to focus on the resolution of structural heterogeneities, i.e., the same semantics is represented by different metamodel concepts, without having to struggle with the intricacies of a certain transformation language. Thereby, in [86] typical mapping situations have been identified, being 1:1 copying, 1:n partitioning, n:1 merging, and 0:1 generating of objects, for which different MOps are provided. In this respect, reuse is leveraged as the proposed MOps are generic in the sense that they abstract from concrete metamodel types since they are typed by the core concepts of current meta-modeling languages like Ecore or MOF (i.e., class, attributes, and references). To further structure the mapping process two steps to specify the actual mapping are proposed.

To exemplify these tow steps and to give a broad overview of the different MOps provided (cf. Fig. 9.16), Fig. 9.17 shows an adapted version of the running example, translating class diagrams to entity-relationship diagrams. In a first step, *composite MOps*, describing mappings between classes are applied, providing an abstract *blackbox-view* (cf. Fig. 9.17). Every composite MOP consists of so-called *kernel MOps*, realizing the composite behavior in terms of a

set of basic building blocks. These kernel MOPs are responsible for resolving structural heterogeneities and therefore, they have to be able to map classes, attributes, and references in all possible combinations and mapping cardinalities (cf. Fig. 9.16). Kernel MOps are provided for copying exactly one object, value, or link from source to target, respectively (denoted as C_2C, A_2A, and R_2R). Moreover, MOps are needed for merging objects, values, and links (denoted as C^n_2C, A^n_2A, and R^n_2R) resolving the structural heterogeneity that concepts in the source metamodel are more fine-grained than in the target metamodel. Finally, MOps are needed for generating a target element without an obvious source element (denoted as 0_2C, 0_2A, and 0_2R) to resolve heterogeneities resulting from expressing the same modeling concept with different meta-modeling concepts – a situation which often occurs in metamodeling practice. In a second step, the composite MOps, which solely describe a mapping between classes at first, have to be refined to also map attributes and references in the so-called *whitebox-view* by the usage of kernel MOps (cf. expanded Copier (b) in Fig. 9.17). Thus, composite MOps describe patterns of heterogeneities, which may contain an arbitrary number of kernel MOps or even other composite MOps. Depending on the specific composite MOp, certain kernel MOps may be automatically derived in order to further ease the mapping specification (cf. [86] for details)

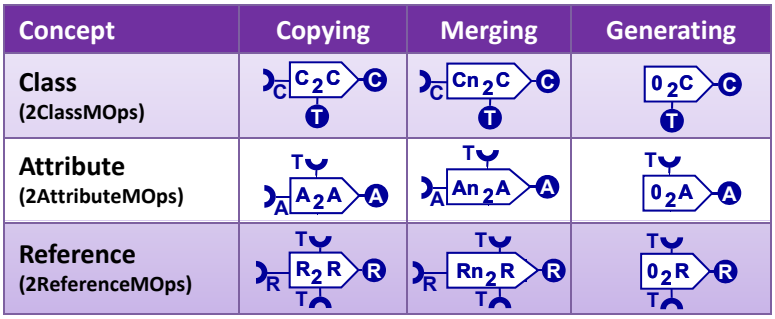


Figure 9.16: Kernel MOps

As a concrete syntax for MOps a subset of the UML 2 component diagram concepts are used enabling the specification of model transformations in a plug & play manner (cf. Fig. 9.16). With this formalism, every MOP is defined as a dedicated component, representing a modular part of the transformation specification which encapsulates an arbitrary complex structure and behavior, providing well-defined interfaces. Every MOP has input ports with required interfaces (left side of the component) as well as output ports with provided interfaces (right side of the component), typed to classes (C), attributes (A), and references (R) (cf. `Copier` (b) in Fig. 9.17). Since there are dependencies between MOps, e.g., a value can only be set after the owning object has been created, MOps dealing with the transformation of classes additionally offer a trace port (T) at the bottom providing *context information*, indicating which target object has been produced from which source object(s). This port can be used by dependent MOps to access context information via required context ports (T). In case of MOps dealing with the mapping of attributes, the corresponding interface is shown via one port on top, or in case of MOps dealing with the mapping of references via two ports, whereby the top port depicts the required source context and the bottom port the required target context (cf. whitebox-view of `Copier` (b) in Fig. 9.17).

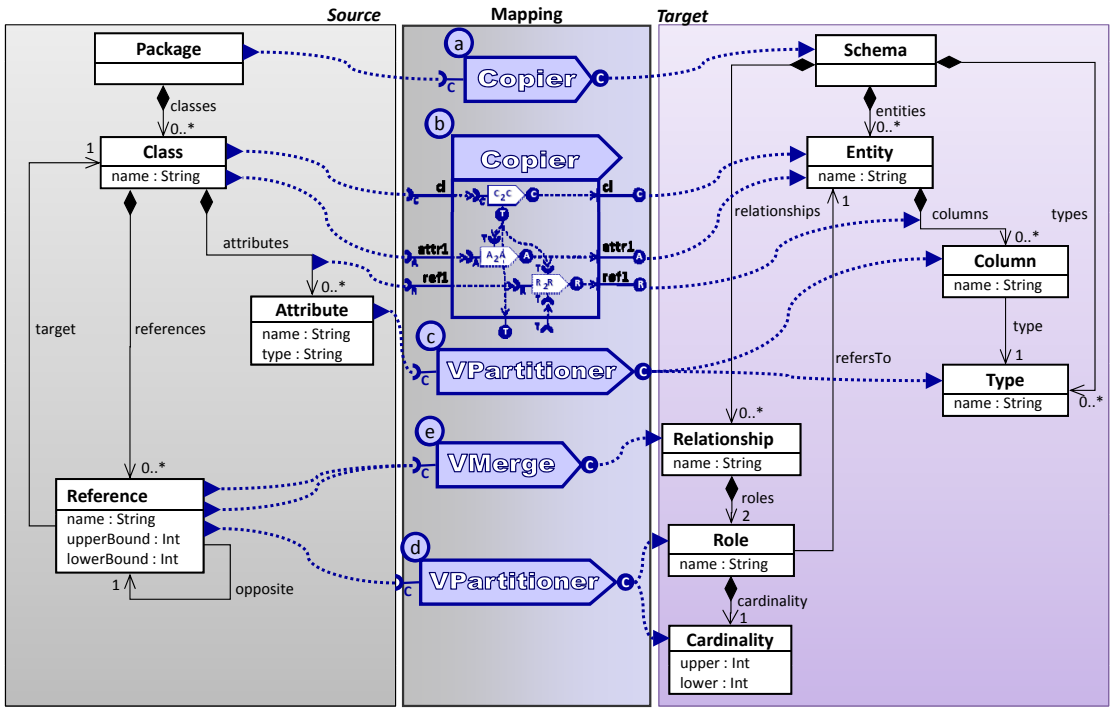


Figure 9.17: Solution of the Running Example

For solving the example, several composite MOps have been applied as can be seen in Fig. 9.17. Table 9.1 presents an overview of the used composite MOps to solve the example as well as their composition of kernel MOps. Thereby, the composition of kernel MOps column in Table 9.1 describes the actual composition of the according composite MOp in EBNF syntax. For example, a `Copier` always consists of a C_2C kernel MOp since it should at least copy the objects from source to target. Additionally, a Copier may incorporate and arbitrary number of kernel MOps dealing with attributes (called `2AttributeMOps`) and references (called `2ReferenceMOps`). For a detailed classification and description of all available kernel as well as composite MOps and their semantics the reader is referred to [86, 166]. To resolve the 1:1 correspondences between `Package` and `Schema` as well as between `Class` and `Entity` in the example, two `Copiers` were applied since for every source object a corresponding target object should be generated (cf. MOps (a) and (b) in Fig. 9.17)). The whitebox-view of the `Copier` (b) thereby shows the mapping of class `Class` to class `Entity` using a C_2C kernel MOp. Moreover, the attribute `Class.name` is mapped to the attribute `Entity.name` by using an A_2A MOp. Finally, the reference `Class.attributes` is mapped to the reference `Entity.columns` using a R_2R MOp. To split the attributes of the class `Reference` to the target classes `Role` and `Cardinality` a `VerticalPartitioner` is applied (cf. MOp (d) in Fig. 9.17). Besides this default behavior, aggregation functionality is sometimes needed as is the case when splitting the `Attribute` concept into the `Column` and `Type` concepts, since a `Type` should only be instantiated for distinct `Attribute.type` values (cf. MOp (c) in Fig. 9.17). Consequently, composite MOps may contain different kinds of kernel MOps, de-

pending on the actual transformation scenario. To finally merge two `Reference` objects to a single `Relationship` object, a `VerticalMerger` is applied (cf. MOp (e) in Fig. 9.17).

Table 9.1: Overview of Composite MOPs used in the Example

Correspondence	MOP	Description	Composition of Kernel MOPs (EBNF)
1:1 - copying	Copier	creates exactly one target object per source object	Copier: C_2C { A_2A | $A^n{}_2A$ | 0_2A | R_2R | $R^n{}_2R$ | 0_2R }
1:n - partitioning	VerticalPartitioner	splits one source object into several target objects	VerticalPartitioner: Copier { ObjectGenerator | Copier }
n:1 - merging	VerticalMerger	merges several source objects to one target object	VerticalMerger: $C^n{}_2C$ { A_2A | $A^n{}_2A$ | 0_2A |R_2R | $R^n{}_2R$ | 0_2R }
0:1 - generating	ObjectGenerator	generates a new target object without corresponding source object	ObjectGenerator: 0_2C { A_2A | $A^n{}_2A$ | 0_2A | R_2R | $R^n{}_2R$ | 0_2R }

9.2.3.2 Compilation of MOps into Transformation Nets

In order to provide an operational semantics for MOps, they need to be mapped to an executable mechanism. In the following it is shown, how the concepts of Transformation Nets may be used to provide semantics for a new transformation language, i.e., MOps in this case.

Compilation Strategy. Every kernel MOps encapsulates a certain operational semantics. In order to encapsulate this operational semantics also in Transformation Nets every kernel MOp is represented in terms of a `Module`. Since composite MOps encapsulate kernel MOps, composite MOps may be represented as `Modules` in Transformation Nets as well. Furthermore, as depicted in Fig. 9.17, every MOp offers ports which are represented by according ports in Transformation Nets. Since these ports provide information that needs to be part of a `Transition` in Transformation Nets, for every port of a MOp additionally an according query or production `Pattern` has to be created. In order to explicate the compilation, in the following the compilation of copying kernel MOps is discussed first, followed by the compilation of merging kernel MOps and concluded with the compilation of generating kernel MOps. Finally, the interaction between MOps is discussed by showing the compilation of the example presented in Fig. 9.17.

Copying Kernel MOps. The operational semantics of copying kernel MOps, i.e., C_2C, A_2A, and R_2R, is to create for every source object, value, or link a corresponding target object, value or link. To represent the operational semantics of the C_2C kernel MOp, a transition comprising a single query `ObjectPattern` and two production patterns (cf. Fig. 9.18) is created. Thereby, the single query `ObjectPattern` matches for source object tokens and the single production object pattern produces an equally colored target token, i.e., the object is copied from source to target. Furthermore, trace information is created, as specified by a corresponding production trace pattern.

This trace information may then be queried by dependent kernel MOps, e.g., by an A_2A, or an R_2R, which both exhibit an according query trace pattern. In this respect, an A_2A MOp exhibits a value query pattern and a context query pattern. The context query pattern is responsible to obtain the object (cf. color on the right), which has been created for the value's owning object (cf. color on the left). Since a newly colored object may have been produced for the original owning object, the value may be in the context of a differently colored object. Therefore, the

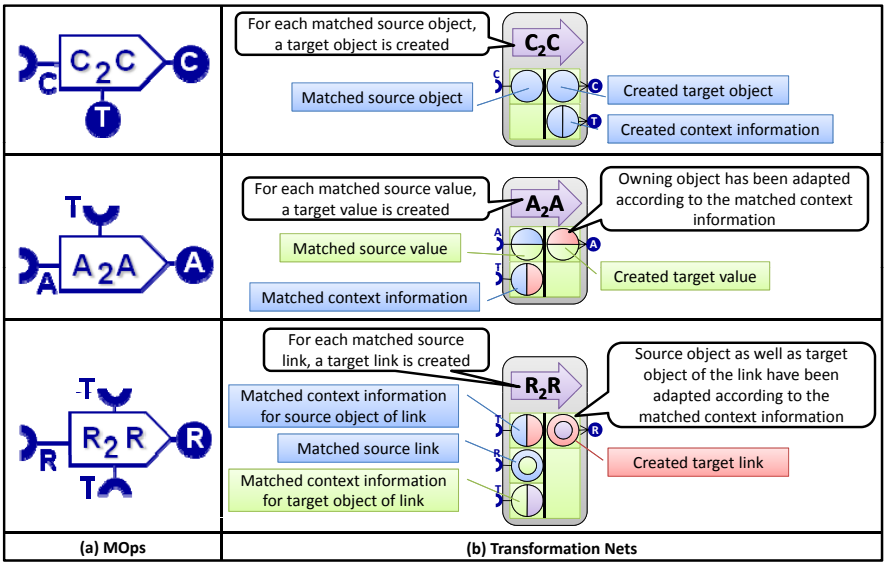


Figure 9.18: Compilation of Copying Kernel MOps

value production pattern comprises as owning object (cf. color on top), the color obtained from the context pattern (cf. color on the right). The same principle is followed by the R_2R MOp, which queries a link instead of a value. Furthermore, since a link requires a source and a target object, an R_2R MOp exhibits two trace query patterns, one pattern for the source and one pattern for the target of the link.

Merging Kernel MOps. After having discussed the operational semantics of copying kernel MOps, in the following the operational semantics of merging kernel MOps is explained. Thereby, merging MOps offer an arbitrary number of according query patterns, i.e., a C_2^nC kernel MOp comprises an arbitrary number of query `ObjectPatterns`, as can be seen in Fig. 9.19. Since in case of C_2^nC MOp a new merged object results, the according production object token exhibits a new color. Furthermore, please note that the left side of the production context pattern contains all the involved source objects. A dependent MOp, e.g., an A_2^nA MOp, is then enabled if the left side of an according query pattern matches to any one of the colors of the left side of the production pattern.

When merging several objects, the question arises under which condition these objects should be merged, i.e., only if there exists a certain link between these objects or if the values of two attributes are equal. In order to specify such conditions, a C_2^nC MOp typically exhibits an according OCL condition which is transferred to the according `Transition` in Transformation Nets. If no condition is specified, the cross-product of the to be merged objects is built. In case of an A_2^nA MOp an according function needs to be defined how to merge the values of the source attributes to a single value of the target attribute which may again be transferred from the MOp's function. This new target value is again represented by means of a newly colored production `ValuePattern`. Finally, in case of an R_2^nR Mop it is assumed that the source link is only merged to a common target link if *all* links between the source and target objects exists - therefore neither a condition nor a function is required.

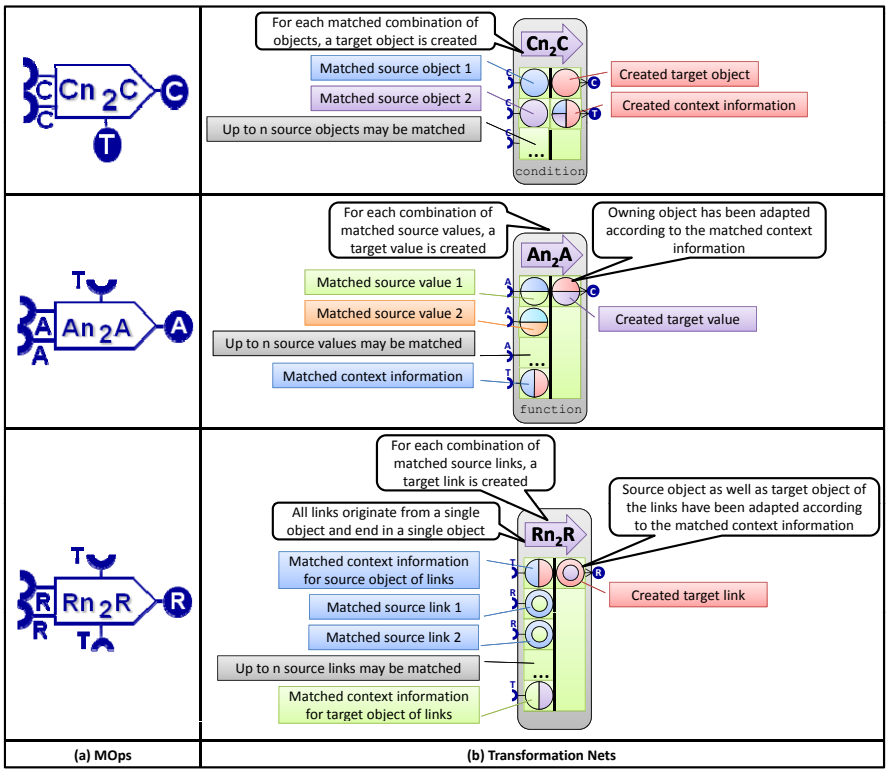


Figure 9.19: Compilation of Merging Kernel MOps

Generating Kernel MOps. As stated above, generating kernel MOps are required to resolve heterogeneities resulting from expressing the same modeling concept with different metamodeling concepts. Thus, kernel MOps are required that are capable to match for a concept, i.e., class, attribute or reference, and produce a different concept thereof. Please note that although numerous combinations are possible, only three representative ones are discussed in detail to present the general idea (for further details the reader is referred to [86]).

To generate an object out of distinct values, as a first generating MOp the A_2C kernel MOp is presented. The according transitions contains a value query token whose distinct flag is set to `true` (cf. Fig. 9.20). In order to produce a target object, the transition exhibits a newly colored object production token. The second production pattern creates context information for dependent MOps. Since context information always comprises objects only, the context production patterns exhibits on the left side the color of the value's owning object. To generate attribute values for objects, a C_2A kernel MOp is provided. The according transitions in Transformation Nets includes a query `ObjectPattern` and a production `ValuePattern` as illustrated in Fig. 9.20. In order to be able to specify the target object to which the generated value belongs to, a query `ContextPattern` is required stating which object has been created from the source object from which the value should be generated. Additionally, the function specified on the MOps to produce a new value has to be transferred to the transition.

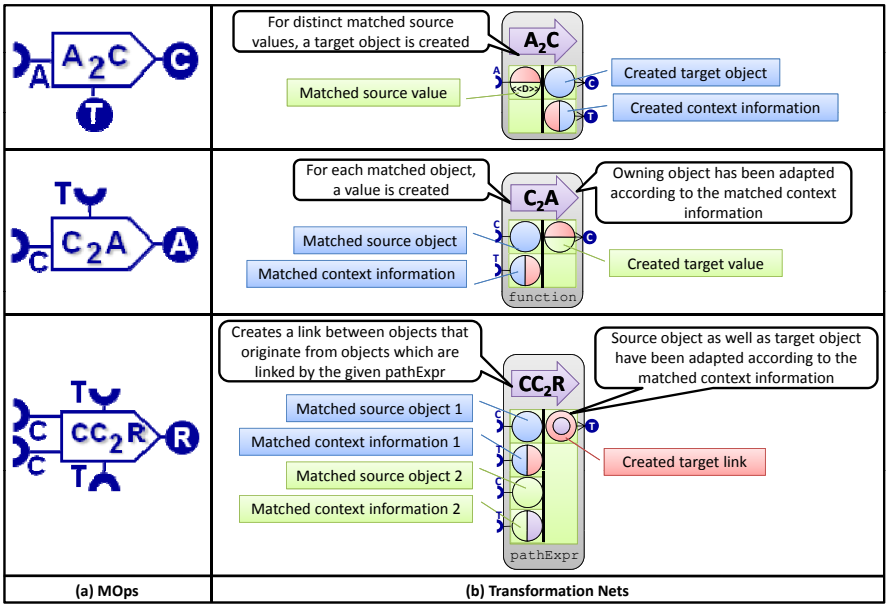


Figure 9.20: Compilation of Generating Kernel MOps

Finally, MOps are required that are capable to generate links in the target model. These MOps require information from which concepts in the source model the links' source and target objects in the target model have been created. In this respect, the CC_2R kernel MOp establishes links between objects that originate from objects which are connected through a corresponding path as determined by the `pathExpr`. The path expression defines how the source and the target object of the to be generated link are related in the source model.This `pathExpr` is taken from the MOps definition and transformed to an according condition on the respective `Transition`. In case of the CC_2R kernel MOp, the transition queries for two objects (the source and target object of the link in the source model) and two context tokens to achieve the according objects in the target model (cf. bottom of Fig. 9.20). The link production pattern then creates links between the objects that have been created from the original objects, i.e., the link pattern exhibits the colors of the queried context information.

Composite MOps. Since composite MOps solely base on kernel MOps, the compilation of composite MOps to Transformation Nets comprises the creation of an according `Module` instance which contains the `Module` instances that are created for the according kernel MOps. Furthermore, according `Ports` need to be added to the `Module` representing the composite MOp. These ports need then to be connected with the according `Ports` of the composite MOp. Additionally, if the transition that represents the operational semantics of a contained kernel MOp contains a production `TracePattern`, an instance of a `TracePlace` is added to the module representing the composite MOp in order to appropriately visualize trace information.

9.2.3.3 Compilation of the Example into Transformation Nets

To exemplify the usage of MOps as well as their compilation, Fig. 9.21 presents an extract of the example presented in Fig. 9.17. In a first step, the involved metamodels are represented as according `Places` in Transformation Nets. When executing a MOp specification a source model is required, which is represented in Transformations Nets by means of according `Tokens`. Afterwards, the mapping specification is translated to Transformation Nets.

When taking a look at the transition that represents the C_2C kernel MOp between the classes `Package` and `Schema` one can see that this MOp is represented by means of an according `Module` and an according `Transition` as discussed above. Since the C_2C is contained within a `Copier` composite MOp, also the `Module` representing the C_2C MOp is contained in a `Module`, which actually represents the `Copier` MOp. To represent trace ports of MOps, trace places are generated which may contain trace tokens indicating which target object has been created from which source object(s) (cf. context places in Fig. 9.21). Since trace information solely contains information about objects, only `Transitions` that generate objects, so-called `2ClassMops`, may be connected to trace places. Context places may be queried by dependent `Transitions` that represents MOps dealing with the creation of attributes, i.e., `2AttributeMOps` and MOps dealing with the creation of references, i.e., `2ReferenceMOps`, in order to obtain the correct context for attribute values and links. Whereas for an attribute value only the context information of the transition representing the `2ClassMops` that is contained within the same composite MOP is required (cf. transition representing the A_2A that is contained within the second module representing the `Copier` MOp), `Transitions` representing `2ReferenceMOps` require a second context for setting the target object of a link. This information is queried from the trace ports of the `2ClassMOp` which translated the target object, i.e, form the C_2C MOp contained in the `Copier` MOp which translates `Class` instances to `Entity` instances.

9.2.4 Comparison to Related Work

The execution engines of current declarative model-to-model transformations act as a black-box to the transformation designer, hiding the operational semantics. For example, the specification of QVT Relations is either translated to the low-level QVT Core language which is then again interpreted by a transformation engine or is directly interpreted by a transforation engine, which is implemented in a low level programming language. This engine is hidden from the transformation designer and thus it remains often unclear why e.g., a certain model element is matched or why a certain relation is executed at a certain point of time. Especially for QVT Relations several so-called translational approaches have been proposed for executing QVT Relations on top of existing technologies in order to explicate the operational semantics. Jouault and Kurtev [74] proposed to execute QVT Relations within the ATL Virtual Machine (ATL VM), by transforming QVT Relations into ATL VM code. Nevertheless, the ATL VM code is on a rather low level of abstraction and no concrete textual syntax may be derived from the VM code. Thus, a transformation designer would have to know the ATL VM code in order to understand the semantics of the translated code. Romeikat et al. [131] transform QVT Relations into the QVT

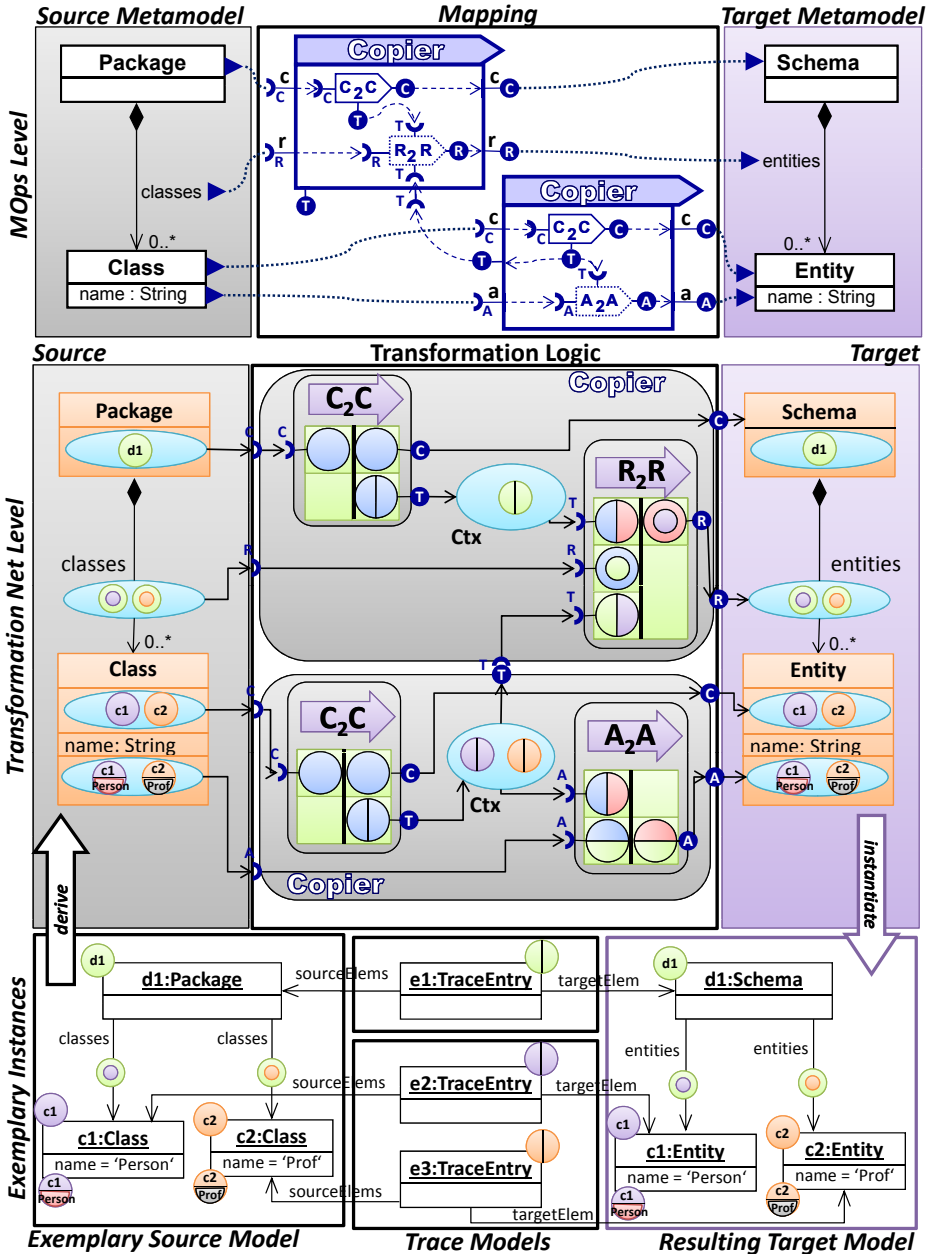


Figure 9.21: Exemplary Compilation of MOps into Transformation Nets

Operational Mappings language and execute the result with tools such as SmartQVT[2]. Since QVT Operational represents an imperative transformation language, an explicit control flow has to be given, e.g., the relations have to be called in a certain order. Thus, possible execution paths

[2]http://smartqvt.elibel.tm.fr/

resulting from non-determinism in the declarative specification might not be accordingly represented. Greenyer and Kindler [53] propose to transform QVT Relations into TGGs. Because QVT Relations and TGGs are conceptually and also syntactically similar, one can remain on the same abstraction level. Nevertheless, since TGGs are not directly executable within existing tools, they have to be translated into executable instructions in order to provide an operational semantics. This semantics is then again hidden from the transformation designer. Another work by de Lara and Guerra [88] proposes to translate QVT Relations directly into CPNs - on the one hand to provide a formal semantics for QVT Relations and on the other hand to verify QVT Relations specifications - pursuing similar ideas to the above presented translation. Nevertheless, the intention of the work of de Lara and Guerra is different to Transformation Nets, since it focuses on QVT Relations only, whereby our Transformation Nets are intended to act as a general runtime model for model-to-model transformations.

In summary, to the best of my knowledge, none of current declarative model-to-model transformation languages provides an explicit runtime model that makes the operational semantics explicit, which could then be used for debugging. In this respect, Transformation Nets provide a runtime model that represents the transformation logic, the metamodels, as well as the respective models involved in a model transformation within a single formalism. Since in Transformation Nets model elements are explicitly represented, it is possible to follow which transformation rule or due to which circumstances a certain model element may or may not be transformed, e.g., in case a certain condition is not fulfilled. Additionally, the interconnections between transformation rules are made explicit using trace information or intermediate places This allows to follow the execution order of certain transformation rules which together form the model transformation.

9.2.5 Summary

In this subsection, the appropriateness of the Transformation Net formalism has been evaluated. Transformation Nets may be used for three different purposes: (i) as a stand alone transformation language, (ii) as a runtime model and execution engine for existing transformation languages, and (iii) as a semantic domain for new transformation languages. Concerning Transformation Nets as a stand alone transformation language, several small extracts have been shown in Chapter 4.

It has to be emphasized, however, that the focus of this thesis was more on the runtime model aspect for existing transformation languages and on providing a semantic domain for new transformation languages. In this respect, it was discussed in detail how the semantics of QVT Relations (implemented by the mediniQVT tool) can be made explicit in Transformation Nets. Furthermore, it was discussed on a conceptual level how graph transformation approaches may be represented. Consequently it was shown that the runtime model is capable to make the semantics of declarative model-to-model transformation languages explicit. Nevertheless, a gap between the actual specification in a certain transformation language and Transformation Nets has been introduced. This is mostly due to the fact that current transformation languages do not provide any means to represent model elements which are nevertheless needed in order to make the matching process of declarative languages explicit. In this respect, a formalism has to be chosen that (i) offers means to represent metamodels, models and transformation logics, (ii)

supports non-determinism, and (iii) provides formal underpinnings to calculate properties about the transformation specification. In this respect, CPNs have been chosen as the most appropriate formalism. In order to hide the low-level details thereof, Transformation Nets as a DSL on top of it have been developed. These adaptations to the domain of model transformations allow for an easier representation of the involved concepts. Furthermore, the visual encoding allows for traceability.

Finally, Transformation Nets are especially useful as a semantic domain for new transformation languages as demonstrated by the compilation of MOps into Transformation Nets. This is since Transformation Nets are not only capable to act as as a semantic domain but also provide several means for debugging. Thus, only a transformation from the actual transformation language to Transformation Nets needs to be provided instead of the tedious implementation of a specific execution engine and according means for debugging.

9.3 Evaluation of Debugging Features

After discussing how Transformation Nets may be used as a runtime model for existing transformation languages, e.g., QVT Relations, graph transformation approaches and MOps, in the following it is shown how the debugging mechanisms of Transformation Nets may be applied. It is presented how the errors contained in the specification of the running example may be detected and how the failures may be fixed using the debugging features provided by Transformation Nets.

9.3.1 Evaluation of Debugging Features of Transformation Nets

In order to evaluate the means provided for debugging, the running example and its solution in QVT Relations is used. When having a look at Fig. 9.10 it can be seen that several modules are connected to the places `Class` and `Table`, which may lead to redundant matches. Furthermore, no modules are connected to the places `Attribute` and `Column`. Thus the question arises how `Attribute` instances get translated into `Column` instances? Furthermore, it can be seen that the trace information that is produced by the `Module` named `SuperAttributeToColumn` is again queried by the same module, representing a recursive call (cf. arc between the outgoing `TracePort` and the incoming `TracePort` in Fig. 9.10 and Fig. 9.22). Thereby the question arises, if the recursive call terminates or not. In order to clarify these questions, simulation-based debugging may be employed as discussed in the following.

In order to debug the QVT Relations code, as a first step, a breakpoint may be attached to the `SuperAttributeToColumn` relation in the QVT code or to the according module in Transformation Nets (cf. Fig. 9.22). Please note that breakpoints which are specified in the QVT Relations code are automatically transferred to breakpoints in Transformation Nets. Fig. 9.22 shows a potential situation when the transition within the `SuperAttributeToColumn` module is enabled first. The transition is only enabled if an according trace token is available in the place `SuperAttribute2Column`, since the relation is not marked as top but called in the when clause of the relation `ClassToTable`. Thus, this transition has to fire before, as can be seen by the tokens in the according target places, i.e., the `Class` instance `c2` has been translated

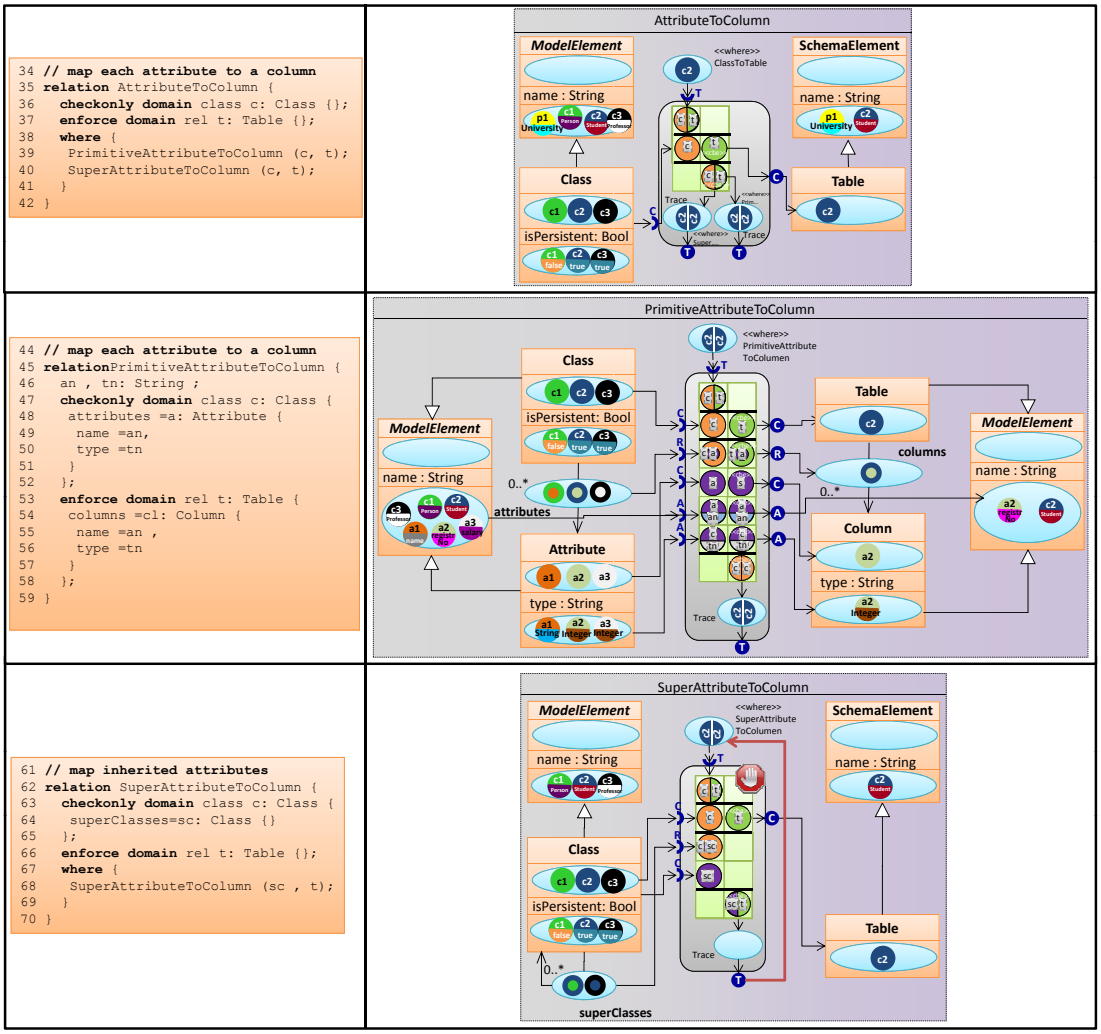


Figure 9.22: Simulation of Erroneous QVT Relations Code

to an according `Table` instance. Since the relation calls two relations in its where clause, two trace places provide the according trace information for the called relations. Since the QVT Relations standard does not explicitly specify any order concerning which called relation should be executed first, a non-deterministic selection is assumed. In the example depicted in Fig. 9.22 first the relation `PrimitiveAttributeToColumn` was executed which can be seen since there exist already tokens in the places `Column`, `Column.type` and `Table.columns`. At this point of time, the relation `SuperAttributeToColumn` may be called and thus the breakpoint stops execution.

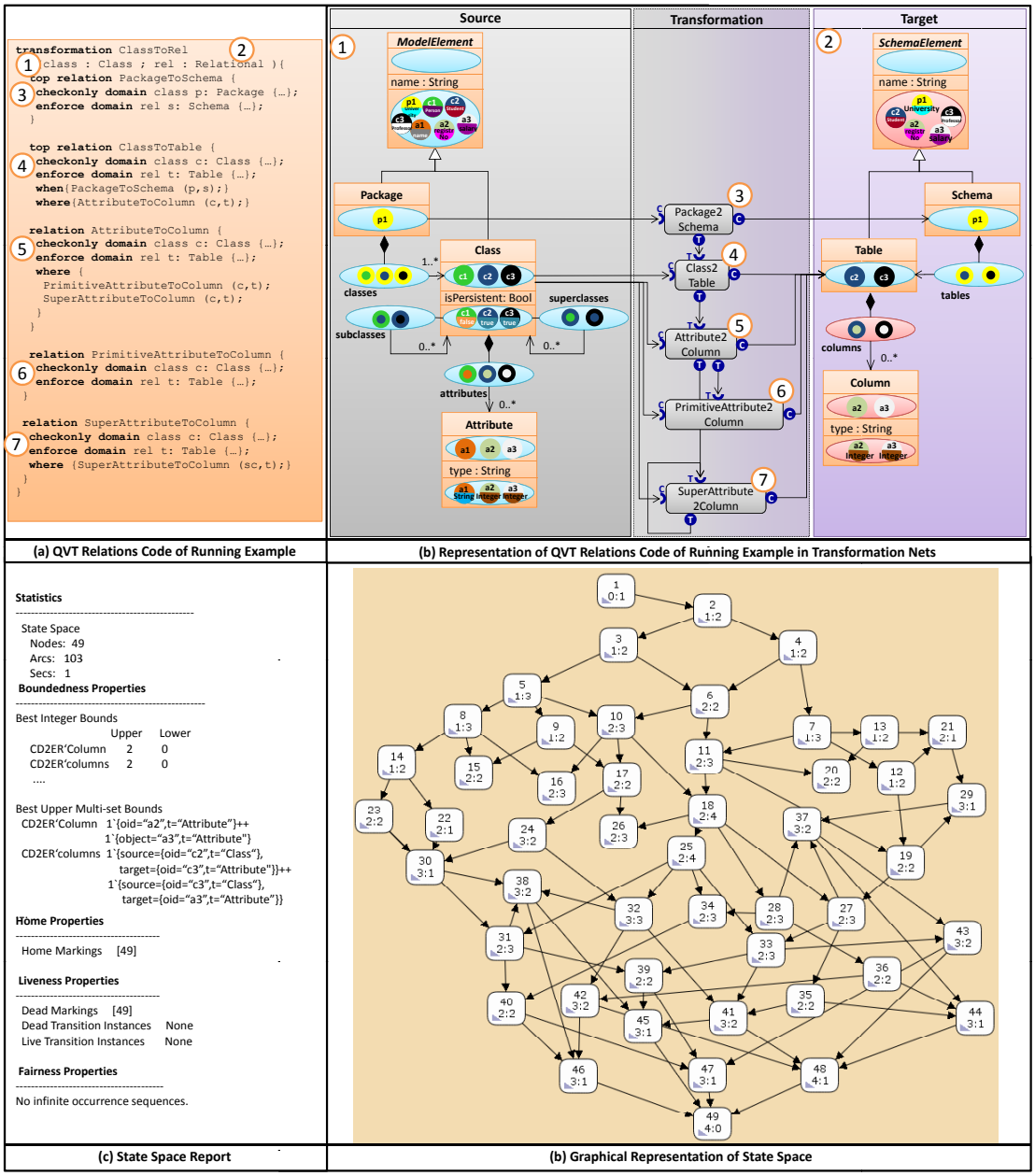
The transformation designer may now fire the according transition in order to stepwise debug the transformation specification. The transition `SuperAttributeToColumn` is enabled since there is an according trace information available and since there exists a link from the class

of the trace information (c2) to a superclass (c1). The found superclass should then be handed to the called relation in the where clause. This is represented in Transformation Nets by the fact that there is an arc between the TracePlace of the relation and the TracePlace that enables the transition, i.e., after firing there would be a second trace token stating that Class c1 is mapped to the Table instance c2. Since the Class instance c1 does not have any further links to super classes, the transition is not enabled any more. Nevertheless, no additional Columns were created, which should be the intention behind the recursive call, according to the stated requirements. In this respect, the origin of the error has been detected by means of simulation-based debugging. The actual bug fixing is discussed below.

9.3.1.1 Evaluation of Property-Based Debugging

The presented methods for property-based debugging (cf. Section 7.4) base on the exploration of the state space. Thus in a first step the state space and according behavioral properties for the running example are calculated and discussed. Furthermore, as stated above, the state space represents all possible paths of execution. A well known difficulty thereby is the so-called state space explosion problem, i.e., the number of states gets to large in order to be represented in memory. There are especially two parameters that influence the state space in CPNs compiled from Transformation Nets being (i) the number of tokens (possible bindings) in the source model and (ii) the number of concurrently enabled transitions. thus mechanisms are demanded that reduce the state space, i.e., means to cluster bindings or to remove concurrency.

Calculation of Behavioral Properties for the Running Example. In order to check properties for the running example, the state space may be calculated. Fig. 9.23(a) shows the according QVT Relations specification which is translated to Transformation Nets in a first step (cf. Fig. 9.23(b)) in order to calculate the state space in a second step (Fig. 9.23(c) and (d)). The state space report tells the transformation designer that the whole state space consists of 49 nodes. This mainly results from the different bindings and due to the fact that several transitions might be enabled concurrently. When investigating the graphical representation of the state space, one sees that in a first step only one transition may fire with one specific binding. This is since exactly one arc originates from the state space node 1, i.e., only the transition PackageToSchema is enabled since there is only one Package instance available in the source model. When further investigating the state space it can be seen that in the second step two possible paths are possible, which represent the two possible bindings for the transition ClassToTable, i.e., only class c2 and c3 are persistent and may thus be transformed to according Table instances. As soon as the transition ClassToTable fires the first time, several paths through the transformation are possible since now several transitions are enabled, i.e., either the transition ClassToTable may fire again but using a different binding, or the transition AttributeToTable may fire. Nevertheless, as can be seen in Fig. 9.23(d) all paths end up at the state space node with the id 49, which has no further successor nodes. Thus, this node is a dead state which always may be reached, i.e., home state, as can be seen also in the state space report in Fig. 9.23(c). In this respect the transformation is confluent with respect to the provided input model. Finally, the question arises, if the transformation is correct. Assuming that the desired target model is available, it can be detected that Column instances as well as their according attributes and links are missing by using boundedness properties. Therefore the corresponding places are highlighted in



(a) QVT Relations Code of Running Example

(b) Representation of QVT Relations Code of Running Example in Transformation Nets

(c) State Space Report

(b) Graphical Representation of State Space

Figure 9.23: Calculation of Properties for Running Example

the Transformation Net (cf. places `Column`, `Column.type`, `SchemaElement.name` and `Class.columns` in Fig. 9.23(b)).

Detection of Non-Confluent Transformation Behavior. As stated in Section 7.4 a transformation is confluent if there exists exactly one dead state which is also a home state. In the example depicted in Fig. 9.24(c) and (d) it can be seen that there exist two different dead states,

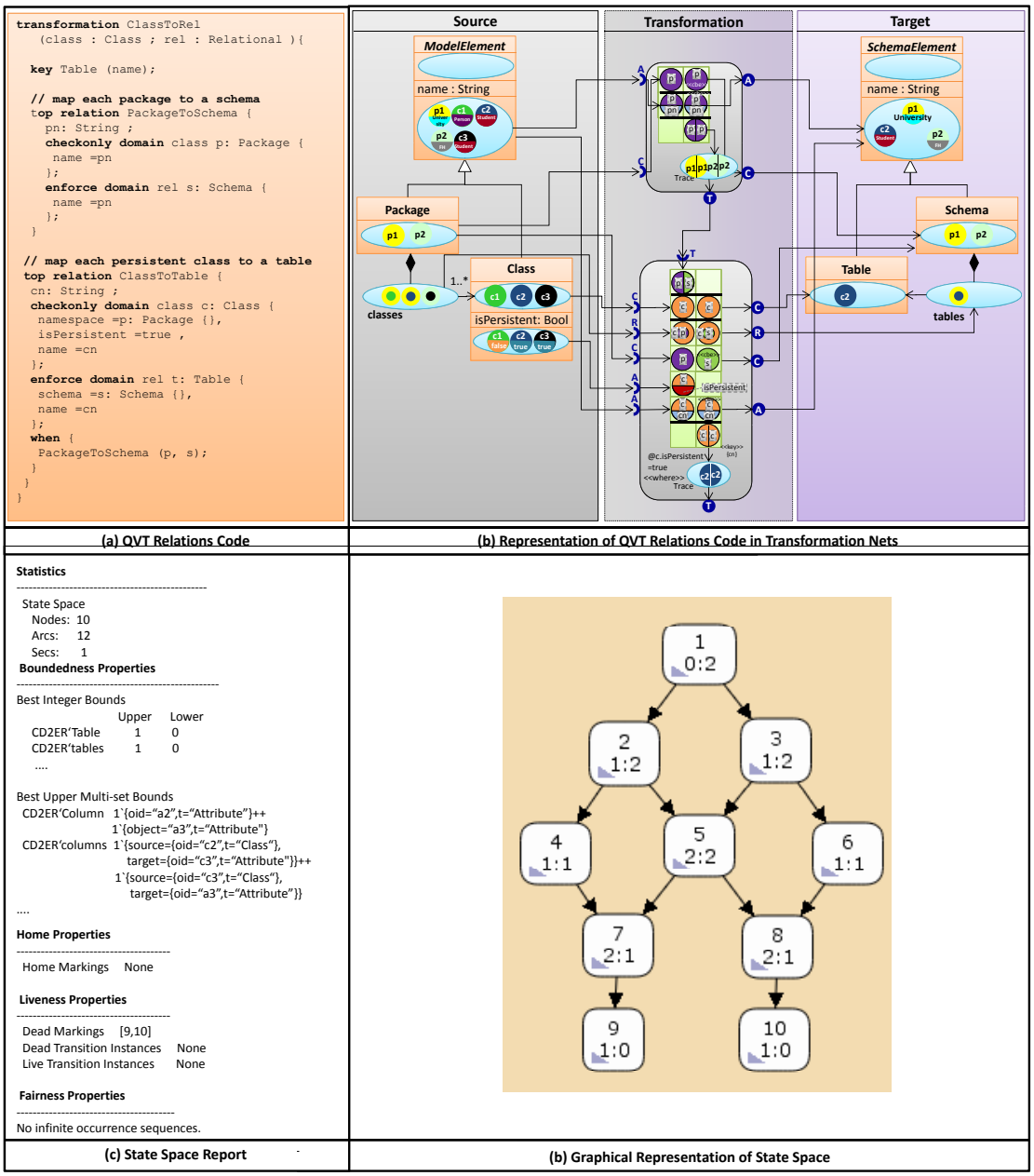


Figure 9.24: Example of Non-Confluent QVT Relations Specification

i.e., state 9 and 10. Therefore, the transformation is non-confluent. When investigating the states and the generated target model it can be seen that always two `Schema` instances get created, which is the intended behavior. Nevertheless, only one `Table` instance results in any case, which is either contained in schema `p1` or `p2`. This is since the specified key in the QVT Re-

lations specification considers only the name of a `Table`. Since the `Class` instances `c2` and `c3` are both named `Student`, only the first, non-deterministically selected `Class` instance gets translated to a `Table` instance. When trying to transform the second instance is matched, no additional `Table` instance is created since already an instance with this key, i.e., with the equal name, exists. If the according `Package` would be considered in the key as well, the transformation would be confluent again (at least for the source model used in the example).

State Space Explosion in Transformation Nets In order to explicate the potential size of a state space Fig. 9.25 shows two simple Transformation Nets and their according state spaces. The Transformation Net in Fig. 9.25(a) simply copies instances of a `Package` to instances of `Schemas`. As can be seen in Fig. 9.25(b), the size of nodes grows exponentially, but the number of states and arcs can still be calculated within a number of seconds. The situation is different when considering the Transformation depicted in Fig. 9.25(c). In this scenario, for every `Class` instance that is contained in an according `Package` instance, a `Table` instance should be generated. Thereby it is assumed that every package contains exactly three `Class` instances. As can be seen in Fig. 9.25(d), the number of states drastically increases when the number of `Package` instances is increased leading to the problem that the state space cannot be calculated any more due to memory restrictions or timing constraints, i.e., the calculation of the state space may last several hours. In this respect, methods for reducing the state space are demanded.

Handling the State Space Explosion Problem. Reduction of state space is still a topic of ongoing research. A major problem thereby is that not every reduction method preserves all properties of the CPN, as stated in [72]. Thus, various different state space reduction methods have been developed that typically allow to exploit certain characteristics only. When applying a concrete method it has to be ensured that the desired properties are preserved. Furthermore, methods can be divided into those that (i) fasten construction or (ii) allow for a more compact representation, i.e., that reduce the numbers of nodes and thus require less memory.

Many of the proposed techniques make use of so-called on-the-fly verification [72]. Thereby the actual verification question and the state space exploration is done relative to the provided verification question, i.e., only those states that are needed for the verification of the desired property are calculated and exploration is stopped as soon as an answer for the desired property has been found. This method is especially common in model checking. Another possibility to reduce the state space is the property preserving symmetry method [72]. The idea behind this method is that systems often exhibit a certain degree of symmetry, i.e., similar components whose identities are interchangeable from a verification point of view. In this respect, so-called equivalence classes are built which represent symmetric markings and binding elements, i.e., each node represents a class of equivalent markings and each arc a class of equivalent binding elements instead of just a specific one. According to [72], these reduced state spaces are typically orders of magnitude smaller than the full state space, but still allow to verify behavioral properties. Nevertheless, the method is only applicable under certain conditions (initial marking must be symmetric, guard expression must be symmetric, and arc expressions must be symmetric - cf. [72] for details) which may be statically checked before calculating the state space. In this respect, the symmetry state space reduction method may be applied for Transformation Nets. Since CPN Tools do currently not include an implementation of the symmetry method, no

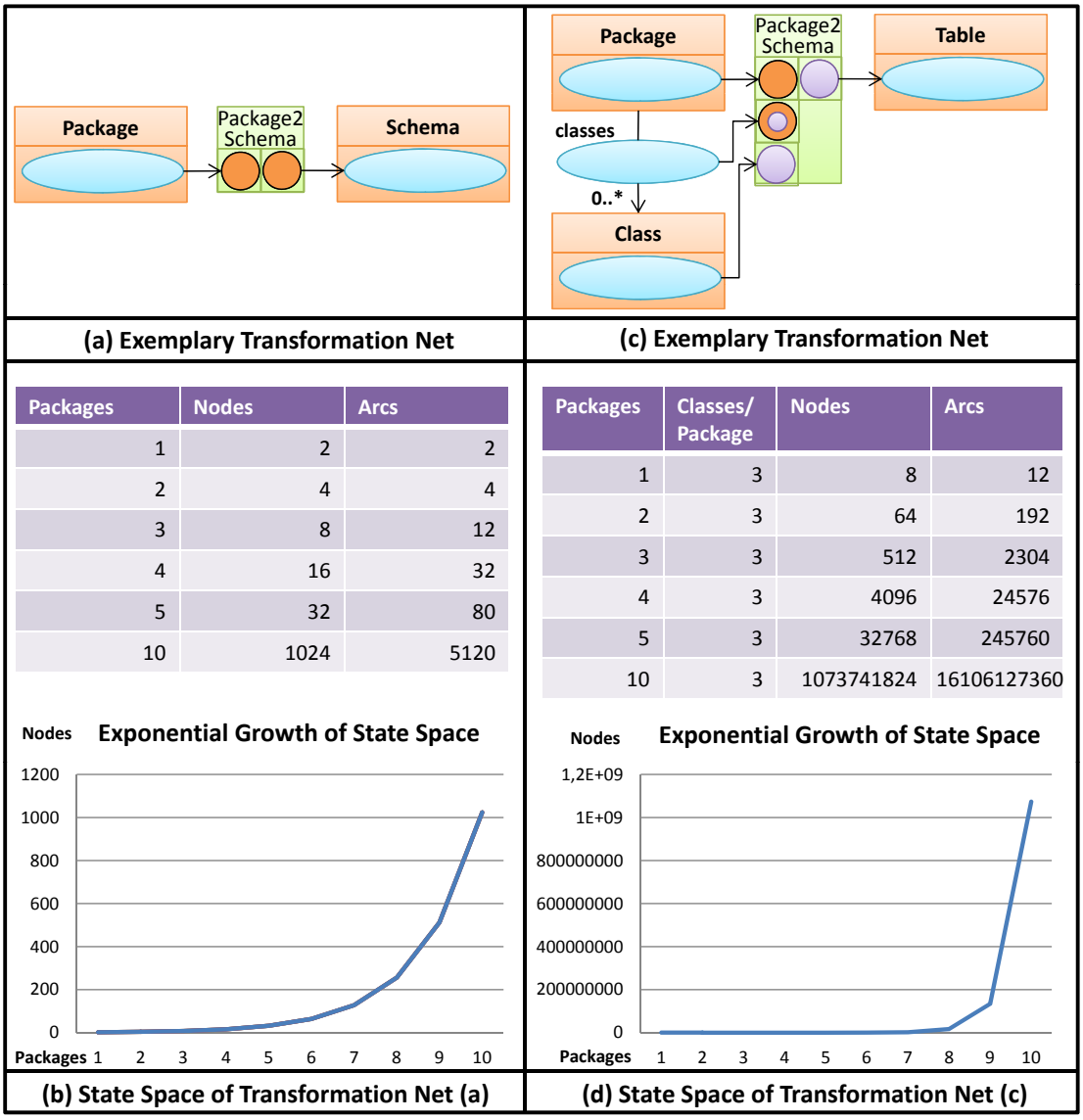


Figure 9.25: Exemplary State Space Calculation

details on the actual reduction of the state space can be presented in this thesis. An implementation of the symmetry method for CPN Tools is thus a point of future work (cf. Chapter 10). Irrespective of the state space reduction method applied, it has to be noted that in general the source models of the Transformation Net under test should be rather small and should focus on a certain scenario in order to be able to calculate the state space in an appropriate amount of time. Furthermore, please note that behavioral properties depend on the actual marking, i.e., source model. This means that if the source model is changed, the according properties need to calculated again.

State Space Reduction for MOPs. A specifics of MOps is that MOps access the source model in a read only manner and the target model in a write only manner, i.e., MOps are not allowed to read from the target model. The trace model may be accessed in a read and write manner, nevertheless it is prohibited to specify conditions on the trace model, i.e., trace patterns may not be included in the condition of a transition. Furthermore, no single production patterns makes use of the check before enforce semantics. Therefore, the transformation results solely depends on the actually matched source model element. As shown above, one source of non-determinism is the selection of one binding out of a number of bindings. Thus, if a marking M has n concurrently enabled bindings, the bindings may be sorted in $n!$ ways, i.e., there are $n!$ paths from a source to a target marking. A possibility to reduce the state space is to order the bindings and to consider only a single path. Thereby, the number of possible bindings may be reduced from $n!$ to n, i.e., the resulting number of paths in the state space is reduced to 1 since the bindings are sequentialized. This sequentialisation preserves all properties due to the above listed specifics of MOps.

Considering the second major factor for state space explosion, MOps also allow to reduce the number of concurrently enabled transitions. A first step in this direction is encoded in the way MOps are specified. This is since `2AttributeMOps` and `2RelationsshipsMOps` may only fire if according trace information is available. Nevertheless, several transitions of MOps of the same type, i.e., `2ClassMOps`, `2AttributeMOps`, or `2RelationsshipsMOps`, may be concurrently enabled. It would be possible to specify a different priority for every transition, whereby the lowest priorities should be assigned to `2ClassMOps` in order to fire first. Additionally, priorities in ascending order should be assigned to `2AttributeMOps` and `2ReferenceMOps`. In this respect it is ensured that first all objects are created, followed by all values and finally all links. Consequently, the possible number of paths through the state space is reduced to exactly one, which has a length of b nodes, whereby b denotes the number of all valid bindings that are possible.

9.3.2 Fixing Bugs

In order to allow the transformation designer to also fix bugs within the Transformation Net formalism, a transformation from Transformation Nets to the actual transformation language is required. As already stated in Section 8.2, as a proof of concept, such transformations are provided for QVT Relations and MOps which will be discussed in the following in order to evaluate the provided means for fixing bugs in Transformation Nets.

Fixing Bugs in Transformation Nets Stemming from a QVT Relations Specification. After finding the origin of a bug, it is possible to adapt the transformation logic during debugging directly in Transformation Nets and propagate the changes back to QVT Relations. Thereby the focus is on bugs in `DomainPatterns`, as fixing those bugs reflects minor changes in QVT Relations and thus can be updated in the debugging environment during the debugging process. As described in Section 9.2 and depicted in Fig. 9.12, the graph of the domain pattern corresponds to a metamodel and is represented in Transformation Nets in terms of `Patterns` contained by `Transitions`. When fixing a bug in Transformation Nets it has to be ensured that the graph remains valid. For example, if a new object, value or link should be queried, it must be ensured that the new Pattern connects to a place which represents a possible new leave

in the graph. Assuming the class `Class` as domain object (cf. example in Fig. 9.12), it would be possible to add a query `LinkPattern` representing the reference `Class.superClass` but it is not possible to query, e.g., for the attribute `Attribute.name` without adding the reference `Class.attributes`, since the attribute `Attribute.name` is no possible leave. If an element of a `DomainPattern` is deleted, all descendent child nodes in the graph must be deleted as well. Nevertheless, the deletion of the domain object is prohibited since this would change the interface of the relation, which might cause also changes in dependent relations, i.e., relations that call this relation in its when or where clause. If such a fundamental modification is necessary, the domain must be changed in the QVT Relations specification and debugging must be restarted. Furthermore, during debugging it should be prohibited to change the metamodels of the transformation as this would result in serious changes in the transformation logic. Although model elements are not represented in QVT Relations, it is possible to add, delete, and edit tokens, which is especially useful to alter the source models during debugging, e.g., by specifying missing attributes.

To hide this complexity from the transformation designer, constraints may be specified which are checked when editing the Transformation Net (as mentioned in Section 8.2). These constraints may either be specified by means of OCL or by implementing a Java method (`ValidateEditOperation`) which is automatically called before the actual operation is executed if the *transformation-based* mode is used in the prototype. In this respect, Table 9.2 shows the possible operations for fixing bugs. The constraints specified in the prototype prohibit any changes on the `PetriNet` itself, i.e., it is not possible to change the involved metamodels, to add transformations (since in QVT Relations it is possible to specify several transformations within a single file), or to delete the transformation specification. As already mentioned, changes concerning the involved metamodels are prohibited whereas changes on the models are allowed. Concerning `Modules` representing `Relations`, only changes are allowed, i.e., contained elements may be changed with the distinction of elements representing the `DomainPattern`. Finally, it is possible to edit, add or delete `Patterns` if a valid digraph results, as described above.

Table 9.2: Possible Operations for Bug Fixing

QVT Concept	TN concept	Edit	Add	Delete
RelationalTransformation	Net	✗	✗	✗
TypedModel	TNPlace	✗	✗	✗
n.a.	Token	✓	✓	✓
Relation	Module	✓	✗	✗
DomainPattern	Transition and ObjectPattern	✗	✗	✗
TemplateExp/PropertyTemplateItem	Pattern	✓	✓	✓

The failure in the QVT Relations specification of the running example is that the wrong relation is called in the where clause of the relation `SuperAttributeToColumn`, i.e., the rela-

tion SuperAttributeToColumn is called instead of the relation AttributeToColumn. To fix this failure in Transformation Nets, the arc stemming from the TracePlace of the module SuperAttributeToColumn needs to be connected to the TracePlace which enables the transition in the module AttributeToColumn (cf. Fig. 9.26). Since this is a valid edit operation, the QVT Relations code is accordingly updated, i.e., the call in the where clause no calls to correct relation AttributeToColumn (cf. Fig. 9.26). When firing the corrected version, the transition in the module SuperAttributeToColumn produces a trace token stating that class c1 (which is the superclass of c2) is translated to table c2. This trace token is then transferred to the trace place in the module AttributeToColumn and therefore enables the according transition again. Since the production ObjectPattern uses the check before enforce semantics, no new table is generated (since an instance c2 already exists). Nevertheless, the newly produced trace information enables the transition in the module PrimitiveAttributeToColumn, which now produces according Column instances for the Attribute instances of the superclass.

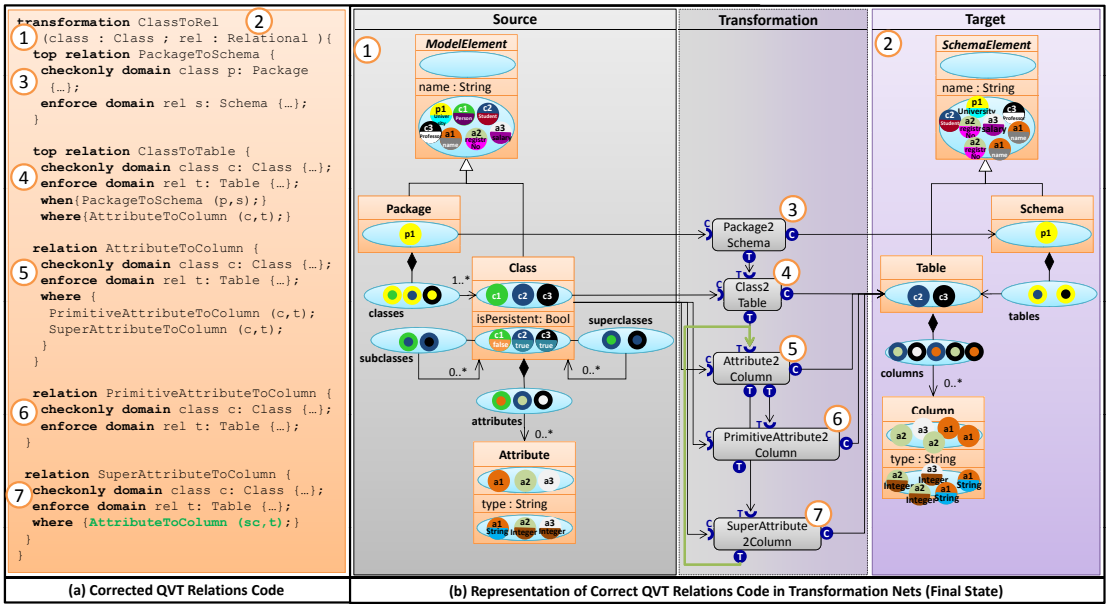


Figure 9.26: Corrected QVT Relations Code of Running Example

Fixing Bugs in Transformation Nets Stemming from a MOps Specification. For fixing bugs in Transformation Nets, which stem from a MOps specification, only a few operations are allowed. Similar to QVT Relations, changes concerning the metamodel, i.e., changes concerning TNPlaces, are prohibited while changes concerning the models, i.e., Tokens are possible. Concerning the actual transformation logic, it is possible to delete instances of Modules, whereby their contained elements, i.e., Ports, Transitions and contained Modules are deleted as well, i.e., either a composite MOp or a kernel MOp may be deleted. Currently, adding elements in Transformation Nets is not allowed (except Arcs between TracePlaces and Ports) since the elements of Transformation Nets represent more fine grained concepts than

those of MOps, i.e., every kernel MOp consists of at least a `Module`, corresponding `Ports` as well as a`Transition` and their contained `Patterns`. Thus, it is more comfortable to alter the according MOps specification and restart debugging. Furthermore, changes concerning `Modules` and `Transitions` are prohibited since the encapsulated operational semantics is fixed by the according MOp. Nevertheless, it is allowed to alter conditions and functions and to change the interplay between MOps, e.g., to edit the `Arcs` between `TracePlaces` and `TracePorts`.

9.3.3 Comparison to Related Work

Table 9.3 compares the debugging features of Transformation Nets to the debugging features of the approaches discussed in Section 2.3. Considering matching support, one can see that Transformation Nets and AGG are the only transformation languages that support the transformation designer during the matching phase of model transformations, i.e., the explicit selection of rules/transitions to fire as well as their according bindings. Comparing the support of breakpoints it can be seen that Transformation Nets support simple as well as conditional breakpoints. Although breakpoints can be seen as a standard debugging support in current transformation languages, Transformation Nets provide more flexible means compared to related approaches, i.e., breakpoints can not only be attached to transitions, but also to modules, places and tokens. Additionally, by means of conditional breakpoints it is possible to define custom breakpoints, e.g., to stop execution if certain places contain a certain number of tokens. A proper support for investigating the current execution state is of major importance for a debugger. Consequently, all of the investigated tools provide support. Nevertheless, most of the tools only provide a tree based visualization of the current variable values. Thus, it is very hard to figure out how the actual target model looks like. Furthermore, the trace model is hidden, even during debugging. This makes it cumbersome to find out which source element has been translated to which target element. Only AGG provides support in the so-called *step mode*, which allows to follow the evolution of the object graph. In order to investigate on operational semantics, debuggers of current model transformation languages provide means to stepwise execute the transformation in a forward direction only (cf. Table 9.3). What sets Transformation Nets apart is that also means for reasoning backwards in time are provided, i.e., it is possible to query a previous execution state by means of (predefined) OCL queries on the runtime model. To actually fix a failure during debugging, little support is provided by current debuggers, i.e., only mediniQVT explicitly allows to change the involved models. In contrast to that, Transformation Nets allow changes in the model, which my be made persistent in the model or not (cf. Section 7.5) and also in the transformation logic itself (cf. above). Finally, since Transformation Nets represent the actual execution of a model transformation again as a model, it is possible to allow for forensic debugging making again use of query based debugging techniques. Furthermore, in combination with PAMOMO contracts, re-enactment is enabled (cf. Section 7.3).

Compared to approaches that support property-based debugging, Transformation Nets do not make abstractions, i.e., the derived CPN fully reflects the the model transformation, which is different compared to e.g., [90]. In this respect Transformation Nets and the underlying CPN bisimulate the according transformation specification. Since the presented properties in [88] base on CPNs as well, similar results may be achieved. Nevertheless, in the paper no bound-

Table 9.3: Debugging Support in Declarative Model Transformation Languages

	Live Debugging									Forensic Debugging
	Selection			Investigation		Dynamics		Adaptations		
	Matching	Breakpoints		State inspection	Visualization of Control Flow	Stepwise Execution	Backwards Reasoning	Model	Logic	
		simple	conditional							
ATL	×	✓	×	✓	✓	✓	×	×	×	×
AGG	✓	×	×	✓	✓	✓	×	~	~	×
Fujaba	×	✓	~ (proposed)	✓	✓	✓	~ (proposed)	~ (proposed)	×	×
GReAT	×	✓	×	✓	✓	✓	×	×	×	×
TGG	~ (proposed)	~ (proposed)	~ (proposed)	~ (proposed)	~ (proposed)	~ (proposed)	×	×	×	~ (proposed)
mediniQVT	×	✓	✓	✓	✓	✓	×	✓	×	×
TNs	✓	✓	✓	✓	✓	✓	✓	✓	~	✓

edness properties are mentioned. Furthermore, it is left open how the results of the properties should be interpreted, i.e., the findings of the properties are not mapped back to the QVT Relations specification.

9.3.4 User study

In order to get a first user feedback if the provided runtime model and the provided means for debugging are appropriate, an admittedly rather small case study has been conducted. Thereby, five master students in software engineering have been selected who attended a course on MDE at the time of writing the thesis. The students had basic knowledge about models, metamodeling and model transformations. Nevertheless, none of the selected students had experiences in QVT Relations nor was familiar with the Transformation Net formalism or CPNs. Three students were selected to debug the corrected running example (cf. Fig. 9.26) which was specified in QVT Relations by using the debugging features provided by mediniQVT. Two students were asked to debug the same transformation but using the features provided by Transformation Nets. The goal of this case study was to report about the transformations tasks. For example question were asked like, what happens if a schema contains no tables or what happens with attributes of subclasses (cf. Table 9.4). The time of debugging was limited to 20 minutes, only.

Results of User Study. The students which used mediniQVT for debugging a QVT Relations specification complained in general that the flow of control was hard to follow. They were not able to clearly state, which model element got translated by which rule and due to which conditions. Another major concern was that they did not see the actual state of the target model. Only one student checked the according variables and tried to reconstruct the model from the in-memory representation. Nevertheless, this was time consuming such that he could not finish debugging in the given time frame. In contrast to that, students debugging with Transformation Nets recognized the explicit visualization of the transitions ready to fire and the possibility to select a certain binding to be helpful to understand the operational semantics. Additionally, they also appreciated the visualization of the models, although they stated that the user needs getting used to the encoding in terms of tokens.

Concerning the question about the operational semantics of the according transformation specification the simple case study showed that although the mediniQVT debugger allows for stepwise execution, the execution was hard to follow, due to the low-level representation, i.e., variable values only. In contrast, the more high-level view of Transformation Nets showed to

be a more appropriate abstraction, although it has to be stated that there is a mismatch between the QVT Relations specification and Transformation Nets as well. In this respect, only two out of three students stated that every package is translated to a schema (actually they counted how often they entered to relation `PackageToSchema` and reasoned that every package is translated) using mediniQVT whereas every student recognized this fact in Transformation Nets (cf. Table 9.4). Both groups were able to figure out that only instances of persistent `Classes` are matched and translated into according `Table` instances. Concerning the interplay between top rules and when clauses, i.e., both relations `ClassToTable` and `PackageToSchema` are marked as top but the relation `ClassToTable` calls the relation `PackageToSchema` in its when clause, only one student recognized that actually first the package has to be generated by using the mediniQVT debugger. In contrast to that, students using Transformation Nets figured out this dependency. Concerning the translation of inherited attributes, only one student recognized that inherited attributes are added to the `Tables` of the according subclass, but all students who used Transformation Nets for debugging figured out that the relation `SuperAttributeToColumn` represents a recursive call. In contrast to that, the recursive call was interpreted right only by one student using mediniQVT. Finally, no user of mediniQVT was able to state the correct number of the generated `Column` instances whereas all users of Transformation Nets stated the right number.

Table 9.4: User Study on Debugging QVT Relations

Questions on Transformations	Given Answers		Correct behavior
	mediniQVT (3 students)	Transformation Nets (2 students)	
What happens if a Package contains no classes?	Are transformed (2) Do not know (1)	Are transformed (2)	every Package is translated to schemas
Which kinds of classes are translated?	Persistent (3)	Persistent (2)	Persistent
Which rule creates the first target object?	ClassToTable (2) PackageToSchema(1)	PackageToSchema (2)	PackageToSchema
What happens with inherited attributes?	Do not know(2) Ignored (1)	Added to table of subclass (1) Do not know (1)	Added to table of subclass
What is the task of the SuperAttributeToColumn rule?	Recursive call (1) Do not know (2)	Recursive call (2)	Recursive call
How many columns result in total?	Do not know (2) Two (1)	Five (3)	Five

9.4 Summary

In summary, this chapter presented an evaluation of the proposed concepts of the thesis. First, an evaluation of the visual declarative language PAMOMO has been conducted by means of case studies. These case studies showed the versatility of the approach by the automated verification of an ATL transformation, a QVT Relation transformation, and the safe execution of a third party transformation (from which the source code is not available). In each case study, the use of different features of PAMOMO was stressed. By comparing PAMOMO to existing approaches the benefits have been highlighted. Second, the runtime model was evaluated by showing how the operational semantics of QVT Relation, graph transformations and MOps can be represented by means of Transformation Net concepts. Consequently, it was shown that the

provided runtime model is able to act as an execution engine for declarative model-to-model transformation languages, providing numerous debugging features. In this respect, third, the debugging features provided by the Transformation Net formalism have again been evaluated by means of case studies and a comparison to related work.

Chapter 10

Conclusion and Future Work

It's the job that's never started
takes longest to finish.

— J.R.R. Tolkien

Contents

This chapter summarizes the contributions of the thesis and gives an outlook on potential lines of future work.

10.1 Conclusion

MDE places models as first-class artifacts throughout the software life cycle, whereby model transformation languages play a vital role. Several kinds of dedicated transformation languages are available, the majority of them favoring declarative, rule based specifications to express relations between source and target models. However, the focus of current model-to-model transformation languages is rather on the implementation phase but they do hardly emphasize phases like requirements specification, testing and debugging. Since the correctness of an automatically generated target model fully depends on the correctness of the specified model transformation, proper support of these phases is heavily demanded. Therefore this thesis focused on providing means to specify requirements for model transformations as well as means to test and debug them.

In order to enable the transformation designer to easily specify the requirements of a model transformation, in the course of this thesis, the language PAMOMO has been adapted for the

specification of requirements. Furthermore, PAMOMO has been implemented in an according tool, called PACO-Checker (cf. ① in Fig. 10.1). By using PAMOMO and PACO-Checker the transformation designer may formalize the requirements in terms of contracts. For this a visual, declarative language is provided, allowing to specify preconditions, postconditions and invariants on transformations. Since the contracts provide a formal semantics, it is possible to reason for inconsistencies between the specified contracts. In order to test if a certain transformation specification fulfills the posed requirements, the contracts are translated to QVT Relations which are then executed in checkonly mode. In this respect it is possible to automatically test the transformation specification against the posed requirements (cf. ③ in Fig. 10.1). Furthermore, a dedicated error trace may be delivered in case a requirement is not fulfilled. This is different to existing approaches that base on OCL were only a boolean answer is provided, i.e., the requirements are fulfilled or not, but no hint is given why the contract failed.

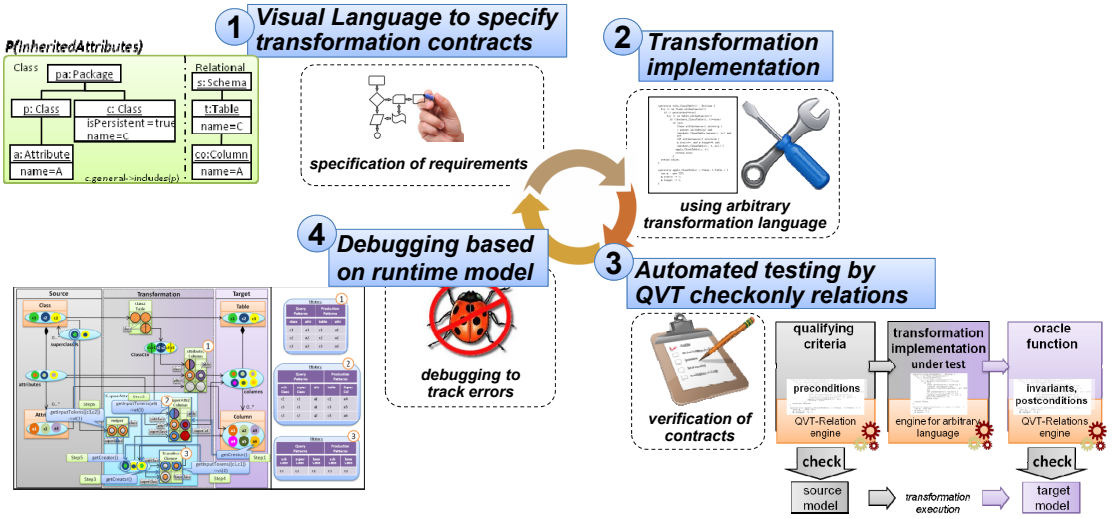


Figure 10.1: Overview on the Contributions of the Thesis

PAMOMO contracts help a transformation designer to observe a certain fact in a transformation, i.e., that a specific requirement is not fulfilled. According to [173], observing facts is a first step in the debugging process only. Additional means are needed to track the origin of a failure and to actually fix it. In order to provide common means for debugging of declarative, rule based transformation languages, Transformation Nets as a runtime model for model-to-model transformations have been introduced making their execution semantics explicit in order to foster debuggability (cf. ④ in Fig. 10.1). Transformation Nets provide a DSL on top of CPNs, which hide the low-level details thereof. Thereby, metamodel elements are represented as places and the according model elements are represented as tokens. The actual transformation logic is specified by a system of transitions.

The Transformation Net formalism provides the basis for several means of debugging in order to find the origin of a failure, comprising means for live-debugging and forensic debugging. First, simulation based debugging has been presented allowing the transformation designer to

stepwise execute a transformation specification. Special emphasis has been put on the proper visualization of the matching phase of the transformation execution. In Transformation Nets it is possible to select a certain transition out of a number of enabled ones (to support non-deterministic rule application) as well as a certain binding. Furthermore, the visualization of the models in terms of tokens representing the current state of the transformation turned out to improve understandability. Besides the common feature of breakpoints, the Transformation Net Debugger additionally allows for more-fine grained conditional breakpoints. To support forensic debugging, so called query-based debugging mechanisms are provided which allow to reason backwards in time by exploiting the runtime model. Since Transformation Nets are defined on the basis of a metamodel, OCL queries can be applied to achieve this task. Finally, since Transformation Nets can be compiled to CPNs, Transformation Nets may make use of the formal properties provided by CPNs, supporting property based debugging. In this respect, behavioral properties may be used in order to check if a transformation specification terminates or is confluent. Since behavioral properties require the calculation of the state space, the well-known state space explosion problem may arise. This is especially the case large if source models which potentially lead to a large number of possible bindings are considered.

After having introduced the concepts of PAMOMO and Transformation Nets, implementation details of the provided prototypes have been discussed. Finally, the proposed contributions have been evaluated by means of comparison to related work as well as case studies. Thereby, the evaluation of the contributions raised potential lines for future work.

10.2 Future Work

The work presented in this thesis leaves several issues open for further research. On the one hand this concerns improvements and extensions of the proposed concepts but also improvements of the presented prototype on the other hand.

10.2.1 Extension of PaMoMo Concepts and Scenarios

Concerning PAMOMO, which allows the specification of visual contracts, as a first major contribution of the thesis, several extensions may be considered.

Traceability between PaMoMo contracts and Transformation Logic. A point for future work is to provide facilities for error location in the transformation implementation. A simple solution would be the manual annotation of the contracts and the according transformation rules, i.e., a transformation rule may be annotated with pre- and postconditions and invariants. In addition, a more complex but also more user friendly approach might be to employ heuristics, used to exploit trace information provided by the execution engines. This trace information might be matched with the error bindings provided by QVT Relations to conclude on the rules causing the error.

From Contracts to Transformation Code. The presented compilation into QVT Relations raises the possibility to use PAMOMO not only as a specification language for contracts, but also as an executable, high level language to specify the actual transformation behavior. This means that the specified invariants may automatically be translated into according transformation rules

of a certain transformation language. Another possibility would be to specify the operational semantics of PAMOMO using Transformation Nets as a semantic domain and execution model. Thus, the specified contracts could not only be made executable but additionally the provided debugging features could be reused also for PAMOMO.

Use of Concrete Syntax. Regarding tool support, the contracts have to be specified in terms of their abstract syntax, i.e., basically the concepts of the metamodels are represented in the tool by means of object diagrams (cf. Fig. 8.2). Nevertheless, typically the transformation designer is used to the concrete syntax only. Therefore, it should be possible to use the concrete syntax of models in PAMOMO specifications as well.

Further Scenarios. The formal semantics of the contracts could not only be used to verify a single transformation but also to analyse the compatibilities of individual transformations in a transformation chain. Additionally, PAMOMO contracts currently target model-to-model transformations only. Consequently, as a point of future work the concepts and the expressive power of PAMOMO contracts may be extended in a way that they may also be used for different scenarios, e.g., in-place transformations.

10.2.2 White-Box Testing of Model Transformations

The presented approach for testing model transformations by means of contracts represents a way of black-box testing a model transformation. This means that the transformation under test is actually not investigated but rather the relationship between the source and the target model. It is for example not tested if a certain rule is executed, or if every possible path of a transformation has been executed at least once. Consequently, if a different input model is used an undesired behavior may occur. In this respect the concept of *dynamic symbolic execution* [151] might be applied also for model transformations (cf. Fig. 10.2). Thereby the specified model transformation is first executed and the path constraints are stored. Path constraints store the evaluation results of certain conditions, e.g., all boolean conditions included in the transition may evaluate to true. If the transformation did not fail, in a second step, the according input model has to be changed in a way that the path constraint is different, e.g., at least one condition

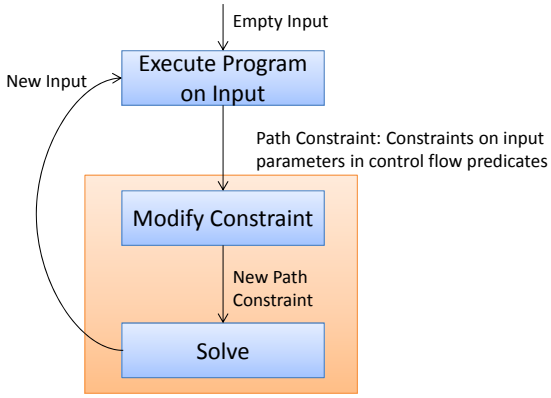


Figure 10.2: Scheme of Dynamic Symbolic Execution (taken from [151])

now evaluates to false. Thus, a different part of the model transformation gets executed. When continuing this process until all possible path constraints have been evaluated, the testing of the transformation specification is finished and the transformation may be assumed to be correct.

10.2.3 Representation of Graph Transformation Languages and Hybrid Transformation Languages in Transformation Nets

As stated in Chapter 9, currently transformations from QVT Relations and MOps are provided in order to allow these transformations languages to reuse the debugging facilities of Transformation Nets. Additionally, the similarity between concepts of Graph Grammars and concepts of Petri Nets have been discussed. However, a transformation from a concrete graph transformation language to Transformation Nets has been conducted on a conceptual level only. As there are several different approaches available for executing graph transformations, e.g., by non-deterministic rule scheduling or by the specification of an external control flow (so-called programmable graph transformation languages). Since the external control flow determines the execution of the rules, this control flow has to be modeled by the according Transformation Net as well. In this respect it has to be first analyzed which methods exist to specify this control flow and how the elements of the control flow can be expressed in terms of Transformation Net concepts in order to support the various approaches for graph transformations.

Furthermore, Transformation Nets currently target declarative model-to-model transformations. Consequently, the question arises how to debug hybrid transformation languages like ATL. Considering the declarative parts only, which may be represented in Transformation Nets, is not sufficient since the imperative parts also influence the operational semantics. Since Transformation Nets are a data-driven formalism, i.e., a transitions fires if a certain configuration of tokens is available, the representation of control sequences seems to be counterintuitive. Nevertheless, for imperative languages, debuggers known from common programming languages might be reused. Thus, a potential solution might be to integrate the debugging features of Transformation Nets and those of common programming languages into a common formalism.

10.2.4 Applying Transformation Nets to Other Scenarios

Currently, the Transformation Net formalism provides a runtime model for declarative model-to-model transformation languages. However, as stated in Subsection 1.1.2, many other scenarios are possible as well. As a first step for future work in this direction it should be investigated how endogenous transformations may be incorporated into the Transformation Net formalism. Another interesting scenario are so-called incremental update transformations since it is sometimes more efficient not to build the target model from scratch but to update it. As the enabling and disabling of transitions in Transformation Nets solely depends on the availability of tokens, i.e., model elements, this mechanisms seems to be promising for incremental updates. First tests with Transformation Nets showed that indeed adding of new, additional source elements may be easily handled. Nevertheless, further investigations are needed concerning updates and deletions of elements in the source model since this might enforce to delete or update elements of the target as well. In this respect, the history and the trace tokens could be used in a similar way as the proposed re-enactment mechanism for forensic debugging in Section 7.3. Nevertheless,

in this approach the histories are updated outside the actual CPN, e.g., by using a Java program. This is not suitable for the actual debugging of incremental updates in transformations since the actual changes and updates in the trace model, the history as well as the target model should be accordingly represented in the Transformation Net itself.

10.2.5 Properties for Model Transformations using Temporal Logics and State Space Reduction Mechanisms

A major constraint concerning the application of behavioral properties to verify the transformation is that the state space typically grows exponentially depending on the number of the input tokens available and the number of concurrently enabled transitions. In this respect, only small transformation specifications may be handled in reasonable time. To overcome these limitations, two different approaches may be followed, namely application of (i) relative state space exploration or (ii) state space reduction mechanisms. Although CPN Tools allow for the specification of LTL formulas which may be used for relative state space exploration, their specification is cumbersome. This is since a model transformation designer is typically not aware of the internals of LTL. Consequently, domain specific verification languages would be needed, that allow the transformation designer to easily specify the according property. For the actual execution, the DSL may then be translated to the according LTL formulas.

Furthermore, CPN Tools does currently not support any reduction mechanism. Thus, as a first step, the according algorithms need to implemented, e.g., an algorithm for the symmetry method, in order to allow for a detailed comparison of different reduction mechanisms and their usefulness in Transformation Nets.

10.2.6 Back Propagation of Bug Fixes

In order to propagate changes in Transformation Nets back to the original transformation specification, an according transformation is required. Nevertheless, the question arises, if this backwards transformation may be derived from the actual forward transformation. A potential solution might be to create an explicit and detailed trace model when specifying the forward transformation. This trace information may then be used in order to derive the backwards transformation, together with a model representing the specifics of a certain transformation language. Additionally, both transformations, i.e., the transformation to Transformation Nets and the transformation back to the original transformation language, should allow for incremental updates in order to be performant.

10.2.7 Improvements on the Prototype and User Studies

The prototype that has been developed during this thesis has been implemented only for a proof of concept but gives room for further improvements. Although the graphical nature of Transformation Nets makes editing and understanding rather intuitive, graphical modeling languages do not scale for large metamodels and models. The current prototype only supports basic mechanisms to scale the graphical representation, e.g., by means of `Modules`. Such package mechanisms may also be used for the representation of metamodels. Furthermore, different layers may

be introduced that allow to hide certain parts of a Transformation Net. Of major importance is also the provision of sophisticated automatic layout algorithms, especially if the Transformation Net is derived from an existing transformation specification, e.g., QVT Relations. If the resulting Transformation Net is not properly layouted, debugging and understandability would be aggravated. This limitations of the current prototype are also a hindering factor to conduct larger user studies in order to evaluate the usefulness of the proposed means for debugging large examples.

Bibliography

[1] Rakesh Agrawal, Linda G. Demichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In Andreas Paepcke, editor, *Proceedings of 6th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications, October 6-11, Phoenix, Arizona*, OOPSLA'91, pages 113–128. ACM, 1991.

[2] Eric Amiel and Eric Dujardin. Supporting Explicit Disambiguation of Multi-Methods. In Pierre Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming, July 8-12, Linz, Austria*, ECOOP'96, pages 167–188. Springer-Verlag, 1996.

[3] Marcel F. Amstel, Christian F. Lange, and Mark G. Brand. Using Metrics for Assessing the Quality of ASF+SDF Model Transformations. In Richard F. Paige, editor, *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations, June 29-30, Zürich, Switzerland*, ICMT '09, pages 239–248. Springer-Verlag, 2009.

[4] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph Transformation for Specification and Programming. *Science of Computer Programming*, 34:1–54, April 1999.

[5] Paolo Atzeni, Gianfor Giorgio, and Paolo Cappellari. Reasoning on Data Models in Schema Translation. In Sven Hartmann and Gabriele Kern-Isberner, editors, *Proceedings of the 5th international Conference on Foundations of Information and Knowledge Systems, February 11-15, Pisa, Italy*, FoIKS'08, pages 158–177. Springer-Verlag, 2008.

[6] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the EASST*, 1, 2007.

[7] Paolo Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars*. PhD thesis, University of Pisa, Department of Informatics, March 2000.

[8] András Balogh, Gábor Bergmann, György Csertán, László Gönczy, Ákos Horváth, István Majzik, András Pataricza, Balázs Polgár, István Ráth, Dániel Varró, and Gergely Varró. Workflow-Driven Tool Integration Using Model Transformations. In *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 224–248. Springer-Verlag, 2010.

[9] András Balogh and Dániel Varró. Advanced model transformation language constructs in the VIATRA2 framework. In Hisham Haddad, editor, *Proceedings of ACM Symposium On Applied Computing, April 23-27, Dijon, France*, SAC '06, pages 1280–1287. ACM, 2006.

[10] 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*, volume 4178 of *Lecture Notes in Computer Science*, pages 306–320. Springer Berlin / Heidelberg, 2006.

[11] Howard Barringer, Bernd Finkbeiner, Yuri Gurevich, and Henny Sipma, editors. *Proceedings of Runtime Verification'05, July 12, Edinburgh, Scotland, UK*, volume 144 of *Electronic Notes in Theoretical Computer Science*, 2005.

[12] Benoit Baudry, Trung Dinh-trong, Jean marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model Transformation Testing Challenges. In Arend Rensink and Jos Warmer, editors, *Proceedings of Integration of Model Driven Development and Model Driven Testing Workshop in conjunction with 2nd European Conference on Model Driven Architecture - Foundations and Applications, July 10-13, Bilbao, Spain*, ECMDA'06. Springer-Verlag, 2006.

[13] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to Systematic Model Transformation Testing. *Communications of the ACM*, 53:139–143, June 2010.

[14] Philip A. Bernstein and Sergey Melnik. Model Management 2.0: Manipulating Richer Mappings. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, June 11 - 14, Beijing, China*, SIGMOD/PODS'07, pages 1–12. ACM, 2007.

[15] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32:38–45, July 1999.

[16] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling*, 4(2):31, 2005.

[17] 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, *Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems, October 1-6, Genova, Italy*, MoDELS'06, pages 440–453. Springer-Verlag, 2006.

[18] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt, editors. *Proceedings of Model Transformations in Practice Workshop in conjunction with 8th International Conference on Model Driven Engineering Languages and Systems, October 1-6, Montego Bay, Jamaica*, (MoDELS'05). Springer-Verlag, 2005.

[19] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, Boston, MA, USA, 1999.

[20] Kurt Bittner and Ian Spence. *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[21] Pierre Bourque, François Robert, Jean-Marc Lavoie, Ansik Lee, Sylvie Trudel, and Timothy C. Lethbridge. Guide to the Software Engineering Body of Knowledge (SWEBOK) and the Software Engineering Education Knowledge (SEEK) - A Preliminary Mapping. In *Proceedings of the 10th International Workshop on Software Technology and Engineering Practice, 6-8 October, Montreal, Quebec, Canada*, STEP '02, page 8. IEEE Computer Society, 2002.

[22] Lionel C. Briand, Yvan Labiche, and Hong Sun. Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code. *Software: Practice and Experience*, 33:637–672, 2003.

[23] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *Proceedings of the 17th International Symposium on Software Reliability Engineering, 7-10 November 2006, Raleigh, North Carolina, USA*, ISSRE'06, pages 85–94. IEEE Computer Society, 2006.

[24] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional, Boston, MA, USA, 2003.

[25] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. A UML/OCL framework for the analysis of graph transformation rules. *Software and System Modeling*, 9(3):335–357, 2010.

[26] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and Validation of Declarative Model-to-Model Transformations through Invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[27] 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, *Proceedings of the 1st international conference on Theory and Practice of Model Transformations, July 1-2, Zürich, Switzerland*, ICMT '08, pages 229–244. Springer-Verlag, 2008.

[28] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *Proceedings of the 1st International Conference on Software Testing Verification and Validation Workshop, April 9-11, Lillehammer, Norway*, number ICST'08, pages 73–80. IEEE Computer Society, 2008.

[29] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. OCL Contracts for the Verification of Model Transformations. *Electronic Communications of the EASST*, 24, 2009.

[30] Eric Cariou, Raphael Marvie, Lionel Seinturier, and Laurence Duchien. OCL for the Specification of Model Transformation Contracts. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *Proceedings of OCL and Model Driven Engineering in conjunction with 7th International Conference on The Unified Modelling Language: Modelling Languages and Applications, October 11-15, Lisbon, Portugal*, UML'04, pages 69–83. Springer-Verlag, 2004.

[31] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming, June 29 - July 3, Utrecht, The Netherlands*, ECOOP'92, pages 33–56. Springer-Verlag, 1992.

[32] Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual Conference on Object-oriented Programming Systems, Languages and Applications, October 21-25, Montreal, Quebec, Canada*, OOPSLA '07, pages 569–588. ACM, 2007.

[33] Minder Chen, Jay F. Nunamaker Jr., and E. Sue Weber. Computer-aided Software Engineering: Present Status and Future Directions. *SIGMIS Database*, 20:7–13, April 1989.

[34] Peter Pin-Shan Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[35] Allan Cheng, Soren Christensen, and Kjeld H. Mortensen. Model Checking Coloured Petri Nets Exploiting Strongly Connected Components. Technical report, Computer Science Department, Aarhus University, 1999.

[36] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[37] Jesús Cuadrado, Esther Guerra, and Juan de Lara. Generic Model Transformations: Write Once, Reuse Everywhere. In Jordi Cabot and Eelco Visser, editors, *Proceedings of 4th International Conference on Model Transformation, June 27-28, Zürich, Switzerland*, ICMT'11, pages 62–77. Springer-Verlag, 2011.

[38] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[39] Andrea Darabos, András Pataricza, and Dániel Varró. Towards Testing the Implementation of Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 211:75–85, 2008.

[40] Eclipse Modeling Framework. The Ecore Metamodel. Available online at `http://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/package-summary.html`. last visited in December 2011.

[41] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*. World Scientific Publishing Co., Inc., 1999.

[42] Marcos Del Fabro and Patrick Valduriez. Towards the Efficient Development of Model Transformations using Model Weaving and Matching Transformations. *Software and Systems Modeling*, 8(3):305–324, July 2009.

[43] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel Matching for Automatic Model Transformation Generation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'08, pages 326–340. Springer-Verlag, 2008.

[44] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, 2001.

[45] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Traon. Qualifying Input Test Data for Model Transformations. *Software and Systems Modeling*, 8:185–203, 2009.

[46] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. In *Proceedings of First International Workshop on Model, Design and Validation in conjunction wiht International Symposium on Software Reliability Engineering, November 2, Rennes, France*, MoDeVVa'04, pages 29–40. IEEE, 2004.

[47] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In Lionel C. Briand and Alexander L. Wolf, editors, *Proceedings of International Conference on Software Engineering in conjunction with 29th International Conference on Software Engineering, May 23-25, Minneapolis, MN, USA*, FOSE'07, pages 37–54. IEEE Computer Society, 2007.

[48] G. Gallasch and L. M. Kristensen. COMMS/CPN: A communication infrastructure for external communication with Design/CPN. In *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, August 29-31, Aarhus, Denmark*, CPN'01, pages 75–91, 2001.

[49] Leif Geiger. Model Level Debugging with Fujaba. In Uwe Aßmann, Jendrik Johannes, and Albert Zündorf, editors, *Proceedings of 6th International Fujaba Days, September 18-19, Dresden, Germany*, pages 23–28, September 2008.

[50] Leif Geiger and Albert Zündorf. eDOBS - Graphical Debugging for Eclipse. *Electronic Communications of the EASST*, 1, 2006.

[51] Pau Giner and Vicente Pelechano. Test-Driven Development of Model Transformations. In Andy Schürr and Bran Selic, editors, *Proceedings of 12th International Conference on Model Driven Engineering Languages and Systems, October 4-9, Denver, CO, USA*, MoDELS'09, pages 748–752. Springer-Verlag, 2009.

[52] Martin Gogolla and Antonio Vallecillo. Tractable Model Transformation Testing. In Robert B. France, Jochen Malte Küster, Behzad Bordbar, and Richard F. Paige, editors, *Proceedings of 7th European Conference on Modelling Foundations and Applications, June 6 - 9, Birmingham, UK*, ECMFA'11, pages 221–235. Springer-Verlag, 2011.

[53] Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems, September 30 - October 5, Nashville, USA*, MoDELS'07, pages 16–30. Springer-Verlag, 2007.

[54] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009. See also `http://www.eclipse.org/modeling/gmp/`.

[55] Esther Guerra, Juan de Lara, Dimitrios Kolovos, Richard Paige, and Osmar dos Santos. Engineering Model Transformations with transML. *Software and Systems Modeling*, pages 1–23, 2011.

[56] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. A Visual Specification Language for Model-to-Model Transformations. In Christopher D. Hundhausen, Emmanuel Pietriga, Paloma Díaz, and Mary Beth Rosson, editors, *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, 21-25 September, Leganés-Madrid, Spain*, pages 119–126, 2010.

[57] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. Inter-modelling: From Theory to Practice. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings of 13th International ConferenceModel Driven Engineering Languages and Systems, October 3-8, Oslo, Norway*, MoDELS'10, pages 376–391. Springer-Verlag, 2010.

[58] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. transML: A Family of Languages to Model Model Transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings of 13th International Conference on Model Driven Engineering Languages and Systems, October 3-8, Oslo, Norway*, MoDELS'10, pages 106–120. Springer-Verlag, 2010.

[59] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 2011. Accepted for Publication.

[60] Laura Haas. Beauty and the Beast: The Theory and Practice of Information Integration. In Thomas Schwentick and Dan Suciu, editors, *Proceedings of 10th International Conference on Database Theory, January 5-7, Edinburgh, UK*, pages 28–43. Springer, 2006.

[61] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamental Informatics*, 26:287–313, June 1996.

[62] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: The Teenage Years. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd international Conference on Very Large Data Bases, September 12-15, Seuol, Korea*, VLDB '06, pages 9–16. ACM, 2006.

[63] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of Semantics? *Computer*, 37:64–72, 2004.

[64] Mary Jean Harrold. Testing: A Roadmap. In *Proceedings of 22nd International Conference on Software Engineering - The Future of Software Engineering Track, June 4-11, Limerick Ireland*, ICSE '00, pages 61–72. ACM, 2000.

[65] 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, *Proceedings of the 1st International Conference on Graph Transformation, October 7-12, Barcelona, Spain*, ICGT '02, pages 161–176. Springer-Verlag, 2002.

[66] Mark T. Hibberd, Michael J. Lawley, and Kerry Raymond. Forensic Debugging of Model Transformations. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems, September 30 - October 5, Nashville, USA*, MoDELS'07, pages 589–604, Nashville, USA, 2007. Springer-Verlag.

[67] Richard Hull and Roger King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Survey*, 19(3):201–260, 1987.

[68] IEEE Computer Society. IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990.

[69] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 13-18, Paris, France*, SIGMOD '04, pages 647–658. ACM, 2004.

[70] Daniel Jackson. *Software Abstractions. Logic, Language, and Analysis*. MIT Press, 2006.

[71] Shawn R. Jeffery, Michael J. Franklin, and Alon Y. Halevy. Pay-as-you-go User Feedback for Dataspace Systems. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 10-12, Vancouver, BC, Canada*, SIGMOD '08, pages 847–860. ACM, 2008.

[72] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets - Modeling and Validation of Concurrent Systems*. Springer, 2009.

[73] 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.

[74] Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. *ACM Symposium on Applied Computing*, 2006.

[75] Yannis Kalfoglou and Marco Schorlemmer. Ontology Mapping: The State of the Art. *Knowledge Engineering Review*, 18:1–31, January 2003.

[76] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *In Preceedings of 9th International Conference on Model Driven Engineering Languages and Systems, October 1-6, Genova, Italy*, MoDELS'06, pages 528–542. Springer-Verlag, 2006.

[77] Gerti Kappel and Michael Schrefl. *Objektorientierte Informationssysteme*. Springer, 1996.

[78] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Example-Based Model-Transformation Testing. *Automated Software Engineering*, 18:199–224, June 2011.

[79] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 3-7, Dubrovnik, Croatia*, pages 285–294. ACM, 2007.

[80] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[81] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Proceedings of 1st International Conference on Model Transformation, July 1-2, Zürich, Switzerland*, ICMT'08, pages 46–60. Springer-Verlag, 2008.

[82] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proceedings of International Workshop on Global Integrated Model Management in conjunction with 28th International Conference on Software Engineering, May 20-28, Shanghai, China*, GaMMa '06, pages 13–20. ACM, 2006.

[83] Alexander Königs. Model Transformation with TGGs. In *Proceedings of Model Transformations in Practice Workshop in cunjunction with 8th International Conference Model Driven Engineering Languages and Systems, October 2-7, Montego Bay, Jamaica*, MoDELS'05. Springer-Verlag, 2005.

[84] Alexander Krasnogolowy. Entwurf und Implementierung eines Debuggers für Story-Diagramme (in German only). Technical report, Master Thesis, Hasso-Plattner-Institut für Softwaresystemtechnik, University of Potsdam, 2010.

[85] Ivan Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 377–393. Springer-Verlag, 2008.

[86] Angelika Kusel. *Reusability in Model Transformations - Resolving Heterogeneites by Composite Mapping Operators*. PhD thesis, Johannes Kepler University Linz, Institute of Bioinformatics, December 2011.

[87] Jochen M. Küster and Mohamed Abd-El-Razik. Validation of Model Transformations: First Experiences Using a White Box Approach. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems, October 1-6, Genova, Italy*, MoDELS'06, pages 193–204. Springer-Verlag, 2006.

[88] Juan de Lara and Esther Guerra. Formal Support for QVT-Relations with Coloured Petri Nets. In Andy Schürr and Bran Selic, editors, *Proceedings of 12th International Conference on Model Driven Engineering Languages and Systems, October 4-9, Denver, CO, USA*, MoDELS'09, pages 256–270. Springer-Verlag, 2009.

[89] Juan de Lara and Hans Vangheluwe. AToM^3: A Tool for Multi-formalism and Metamodelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Proceedings of 5th International ConferenceFundamental Approaches to Software Engineering in conjunction with Joint European Conferences on Theory and Practice of Software, April 8-12, Grenoble, France*, FASE'02, pages 174–188. Springer-Verlag, 2002.

[90] Juan de Lara and Hans Vangheluwe. Automating the Transformation-Based Analysis of Visual Languages. *Formal Aspects of Computing*, 21, Mai 2009.

[91] Michael Lawley and Jim Steel. Practical Declarative Model Transformation with Tefkat. In *Proceedings of MoDELS Satellite Events on 8th International Conference Model Driven Engineering Languages and Systems, October 2-7, Montego Bay, Jamaica*, MoDELS'06, pages 139–150. Springer-Verlag, 2006.

[92] Yves Le Traon, Benoit Baudry, and Jean-Marc Jezequel. Design by Contract to Improve Software Vigilance. *IEEE Transactions on Software Engineering*, 32:571–586, 2006.

[93] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the Design of JML Accommodates both Runtime Assertion Checking and Formal Verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

[94] Frank Legler and Felix Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. In Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *Proceedings of the GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web, March 7-9, 2007, Aachen, Germany*, BTW'07, pages 449–464. GI-LNI, 2007.

[95] Grzegorz Lehmann, Marco Blumendorf, Frank Trollmann, and Sahin Albayrak. Meta-Modeling Runtime Models. In Dorina C. Petriu, Nicolas Rouquette, and Oystein Haugen, editors, *Proceedings of 13th International Conference on Model Driven Engineering Languages and Systems, October 3-8, Oslo, Norway*, MoDELS'10, pages 209–223. Springer-Verlag, 2011.

[96] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical Object-Oriented Back-in-Time Debugging. In Jan Vitek, editor, *Proceedings of the 22nd European Conference on Object-Oriented Programming, July 7-11, Paphos, Cyprus*, ECOOP '08, pages 592–615. Springer-Verlag, 2008.

[97] Yuehua Lin, Jing Zhang, and Jeff Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of Best Practices for Model-Driven Software Development Workshop in conjunction with 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 24-28, Vancouver, BC, Canada*, OOPSLA'04. ACM, 2004.

[98] Yuehua Lin, Jing Zhang, and Jeff Gray. A Testing Framework for Model Transformations. In *Model-Driven Software Development - Research and Practice in Software Engineering*, pages 219–236. Springer-Verlag, 2005.

[99] Barbara Liskov and Jeannette M. Wing. A New Definition of the Subtype Relation. In Oscar Nierstrasz, editor, *Proceedings of 7th European Conference Object-Oriented Programming, July 26-30, Kaiserslautern, Germany*, ECOOP'93, pages 118–141. Springer-Verlag, 1993.

[100] Haohai Ma, Weizhong Shao, Lu Zhang, Zhiyi Ma, and Yanbing Jiang. Applying OO metrics to assess UML meta-models. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *Proceedings of 7th International Conference on The Unified Modelling Language: Modelling Languages and Applications, October 11-15, Lisbon, Portugal*, UML'04, pages 12–26. Springer-Verlag, 2004.

[101] Raphael Mannadiar and Hans Vangheluwe. Debugging in Domain-Specific Modelling. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Proceedings of 4th International Conference on Software Language Engineering, July 3-6, Brage, Portugal*, volume 6563 of *SLE'11*, pages 276–285. Springer-Verlag, 2011.

[102] Shahar Maoz. Using Model-Based Traces as Runtime Models. *Computer*, 42:28–36, October 2009.

[103] Maria Maximova, Hartmut Ehrig, and Claudia Ermel. Formal Relationship between Petri Net and Graph Transformation Systems based on Functors between $\mathcal{M}$-adhesive Categories. *ECEASST*, 40, 2011.

[104] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model Driven Architecture*. Addison-Wesley Professional, Boston, MA, USA, 2004.

[105] Bertrand Meyer. Applying Design by Contract. *Computer*, 25:40–51, October 1992.

[106] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, 1990.

[107] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of 38th Design Automation Conference, June 18-22, Las Vegas, NV, USA*, DAC'01, pages 530–535. ACM, 2001.

[108] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems, October 1-6, Genova, Italy*, MoDELS'06, pages 589–603. Springer-Verlag, 2006.

[109] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Model Transformation Testing: Oracle Issue. In *Proceedings of the IEEE International Conference on Software Testing Verification and Validation, April 9-11, Lillehammer, Norway*, ICST'08, pages 105–112. IEEE Computer Society, 2008.

[110] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In S. Briand and Kent L., editors, *Proceedings of 8th International Conference on Model Driven Engineering Languages and Systems, October 1-6, Montego Bay, Jamaica*, MoDELS'05, pages 264–278. Springer-Verlag, 2005.

[111] Nataliya Mulyar and Wil M. P. van der Aalst. Towards a pattern language for Colored Petri Nets. In Kurt Jensen, editor, *Proceedings of 6th Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, October 24-26, Aarhus, Denmark*, pages 39–58, 2005.

[112] Tadao Murata. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[113] Oscar Nierstrasz. Synchronizing Models and Code (Invited Talk). In Judith Bishop and Antonio Vallecillo, editors, *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns, June 27 - July 1, Zürich, Switzerland*, TOOLS'11, pages 1–1. Springer-Verlag, 2011.

[114] Natalya F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Record*, 33:65–70, December 2004.

[115] Object Management Group. OCL Specification Version 2.0. http://www.omg.org/docs/ptc/05-06-06.pdf, 2005.

[116] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/spec/QVT/1.1/Beta2/PDF/, 2009.

[117] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the 11th International Conference on Data Engineering, March 6-10, Taipei, Taiwan*, ICDE '95, pages 251–260. IEEE Computer Society, 1995.

[118] Carl Adam Petri. Fundamentals of a Theory of Asynchronous Information Flow. In Cicely M. Popplewell, editor, *Proceedings of IFIP Congress, August 27-September 1, Munich, Germany*, pages 386–390, 1962.

[119] Detlef Plump. Termination of Graph Rewriting is Undecidable. *Fundamental Informatics*, 33(2), 1998.

[120] Guillaume Pothier and Eric Tanter. Back to the Future: Omniscient Debugging. *IEEE Software*, 26:78–85, 2009.

[121] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: A Visual Language for Explicit Schema Mappings. In *Proceedings of the 24th International Conference on Data Engineering, April 7-12, 2008, Cancún, México*, ICDE'08, pages 30–39, 2008.

[122] Erhard Rahm and Philip A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001.

[123] Rodrigo Ramos, Olivier Barais, and Jean-Marc Jézéquel. Matching Model-Snippets. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems, September 30 - October 5, Nashville, USA,*, MoDEL'07, pages 121–135. Springer-Verlag, 2007.

[124] Wolfgang Reisig. *Petri Nets, An Introduction*. Springer-Verlag, 1985.

[125] Thomas Reiter. *T.R.O.P.I.C.: Transformations On Petri Nets In Color*. PhD thesis, Johannes Kepler University Linz, Faculty of Bioinformatics, Februar 2008.

[126] Thomas Reiter, Manuel Wimmer, and Horst Kargl. Towards a Runtime Model based on Colored Petri-Nets for the Execution of Model Transformations. In *In Proceedings of 3rd Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD, July 30 - August 3, Berlin, Germany*, pages 19–23, 2007.

[127] Arend Rensink. Towards Model Checking Graph Grammars. In M. Leuschel, S. Gruner, and S. Lo Presti, editors, *In Proceedings of Workshop on Automated Verification of Critical Systems, April 2-3, Sourthampton, Great Britain*, AVoCS'03, pages 150–160. University of Southampton, 2003.

[128] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Proceedings of Applications of Graph Transformations with Industrial Relevance*, AGTIVE'04, pages 479–485. Springer-Verlag, 2004.

[129] José Eduardo Rivera, Esther Guerra, Juan Lara, and Antonio Vallecillo. Software Language Engineering. chapter Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude, pages 54–73. Springer-Verlag, 2009.

[130] Suzanne Robertson and James Robertson. *Mastering the requirements process*. ACM Press/Addison-Wesley Publishing Co., 1999.

[131] Raphael Romeikat, Stephan Roser, Pascal Müllender, and Bernhard Bauer. Translation of QVT Relations into QVT Operational Mappings. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Proceedings of 1st International Conference on Theory and Practice of Model Transformations, July 1-2, Zürich, Switzerland*, ICMT'08, pages 137–151. Springer-Verlag, 2008.

[132] Marko Sakkinen. Disciplined Inheritance. In S. Cook, editor, *Proceedings of 3rd European Conference on Object-Oriented Programming, July 10-14, Nottingham, UK*, ECOOP'89, pages 39–56. Cambridge University Press, 1989.

[133] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39, February 2006.

[134] Johannes Schönböck. Transformation Nets: A Runtime Model for Transformation Languages. In *Proceedings of the Doctoral Symposium at Model Driven Engineering Languages and Systems, October 4-9, Denver, CO, USA*, MoDELS'09. School of Computing, Queen's University, 2009.

[135] Johannes Schönböck, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Catch Me If You Can - Debugging Support for Model Transformations. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009. Reports and Revised Selected Papers*, LNCS 6002, pages 5–20. Springer-Verlag, 2010.

[136] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Proceedings of the*

20th International Workshop on Graph-Theoretic Concepts in Computer Science, June 16-18, Herrsching, Germany, WG'94, pages 151–163. Springer-Verlag, 1994.

[137] Mirko Seifert and Stefan Katscher. Debugging Triple Graph Grammar-based Model Transformations. In Uwe Aßmann, Jendrik Johannes, and Albert Zündorf, editors, *Proceedings of 6th International Fujaba Days, September, Dresden, Germany*, pages 23–28, September 2008.

[138] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software.*, 20:19–25, September 2003.

[139] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *Proceedings of 1st International Conference on Software Testing, Verification, and Validation, April 9-11, Lillehammer, Norway*, ICST'08, pages 328–337. IEEE Computer Society, 2008.

[140] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic Model Generation Strategies for Model Transformation Testing. In Richard Paige, editor, *Theory and Practice of Model Transformations*, LNCS, pages 148–164. Springer-Verlag, 2009.

[141] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Survey*, 22(3):183–236, 1990.

[142] Nan C. Shu, Barron C. Housel, Robert W. Taylor, Sakti P. Ghosh, and Vincent Y. Lum. EXPRESS: A Data EXtraction, Processing, and Restructuring System. *ACM Transactions on Database Systems*, 2:134–174, June 1977.

[143] J. Michael Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, 1989.

[144] Perdita Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, September 30 - October 5, Nashville, USA*, MoDELS'07, pages 1–15. Springer-Verlag, 2007.

[145] Perdita Stevens. A Simple Game-Theoretic Approach to Checkonly QVT Relations. In Richard F. Paige, editor, *Proceedings of 2nd International Conference on Model Transformation Theory and Practice of Model Transformations, June 29-30, Zurich, Switzerland*, ICMT'09, pages 165–180. Springer-Verlag, 2009.

[146] Michael Strommer and Manuel Wimmer. A Framework for Model Transformation By-Example: Concepts and Tool Support. In Bertrand Meyer and Richard Paige, editors, *Proceedings of 46th International Conference on Objects, Components, Models and Patterns, June 30 - July 4, Zurich, Switzerland*, TOOLS, pages 372–391. Springer-Verlag, 2008.

[147] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Proceedings of Applications of Graph Transformations with Industrial Relevance*, AGTIVE'04, pages 446–453. Springer-Verlag, 2004.

[148] Antero Taivalsaari. On the Notion of Inheritance. *ACM Computing Survey*, 28(3):438–479, 1996.

[149] TATA Research Development and Design. ModelMorf. `http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`. Last accessed: November 2011.

[150] Frank Tip. A survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[151] Frank Tip. Finding and Fixing Bugs in Web Applications (Invited Talk). In Judith Bishop and Antonio Vallecillo, editors, *Proceedings of 49th International Conference on Objects, Models, Components, Patterns, June 28-30, Zurich, Switzerland*, TOOLS'11, page 2. Springer-Verlag, 2011.

[152] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of 5th European Conference on Model Driven Architecture - Foundations and Applications, June 23-26, Enschede, The Netherlands*, ECMDA-FA'09, pages 18–33. Springer-Verlag, 2009.

[153] Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Towards Dynamic Backward Slicing of Model Transformations. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *Proceedings of 26th IEEE/ACM Conference on Automated Softweare Engineering, November 6-10, Lawrence, Kansas,USA*, ASE'11, page 4. IEEE Computer Society, 2011.

[154] Jeffrey D. Ullman. *Elements of ML programming (ML97 ed.)*. Prentice-Hall, Inc., 1998.

[155] Antti Valmari. The State Explosion Problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *In Proceedings of Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, based on the Advanced Course on Petri Nets, September, Dagstuhl, Germany*, pages 429–528, London, UK, 1998. Springer-Verlag.

[156] Dániel Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *Software and Systems Modelling*, 3(2):85–113, 2003.

[157] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination Analysis of Model Transformation by Petri Nets. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Proceedings of 3rd International Conference on Graph Transformations, September 17-23, Natal, Rio Grande do Norte, Brazil*, ICGT'06, pages 260–274. Springer-Verlag, 2006.

[158] Philip Wadler. Why no one uses Functional Languages. *SIGPLAN Notices*, 33(8):23–27, 1998.

[159] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module Superimposition: A Composition Technique for Rule-Based Model Transformation Languages. *Software and Systems Modelling*, 9:285–309, 2010.

[160] Robert Wagner. Developing Model Transformations with Fujaba. In Holger Giese and Bernhard Westfechtel, editors, *Proceedings of the 4th International Fujaba Days, September, Bayreuth, Germany*, pages 79–82, 2006.

[161] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. *Petri Net Technology for Communication-Based Systems*, pages 124–144, 2003.

[162] Mark Weiser. Program Slicing. In *Proceedings of the 5th international Conference on Software Engineering, March 9-12, San Diego, California, USA*, ICSE '81, pages 439–449. IEEE Computer Society, 1981.

[163] Michael Westergaard, Sami Evangelista, and Lars Michael Kristensen. ASAP: An Extensible Platform for State Space Analysis. In Giuliana Franceschinis and Karsten Wolf, editors, *Proceedings of 30th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency, June 22-26, Paris, France*, PETRI NETS '09, pages 303–312. Springer-Verlag, 2009.

[164] Michael Westergaard and Lars Michael Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In Giuliana Franceschinis and Karsten Wolf, editors, *Proceedings of 30th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency, June 22-26, Paris, France*, PETRI NETS '09, pages 313–322. Springer-Verlag, 2009.

[165] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger. Right or Wrong? - Verification of Model Transformations using Colored Petri Nets. In M. Rossi, J. Sprinkle, J. Gray, and J.-P. Tolvanen, editors, *Proceedings of 9th Workshop on Domain-Specific Modeling, held in conjunction with 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 25-26, Orlando, USA*, OOPSLA'09, 2009.

[166] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In Laurence Tratt and Martin Gogolla, editors, *Proceedings of 3rd International Conference on Theory and Practice of Model Transformations, June 28-July 2, Malaga, Spain*, ICMT'10, pages 260–275. Springer-Verlag, 2010.

[167] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris Kolovos, Richard Paige, Marius Lauder, Andy Schürr,

and Dennis Wagelaar. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In Jordi Cabot and Eelco Visser, editors, *Proceedings of the 4th International Conference on Model Transformations, June 27-28, Zurich, Switzerland*, ICMT'11, pages 31–46. Springer-Verlag, 2011.

[168] Manuel Wimmer, Gerti Kappel, Johannes Schönböck, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. A Petri Net based Debugging Environment for QVT Relations. In *Proceedings of the 24th International Conference on Automated Software Engineering, November 16-20, Auckland, New Zealand*, ASE'09, pages 1–12. IEEE Computer Society, 2009.

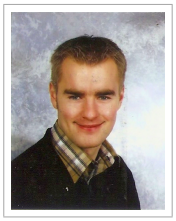
[169] Manuel Wimmer, Gerti Kappel, Johannes Schönböck, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. TROPIC: A Framework for Model Transformations on Petri Nets in Color. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 25-29, 2009, Orlando, Florida, USA*, OOPSLA 2009, pages 783–784. ACM, 2009.

[170] Manuel Wimmer, Angelika Kusel, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. Lost in Translation? Transformation Nets to the Rescue! In Jianhua Yang, Athula Ginige, Heinrich C. Mayr, and Ralf-Detlef Kutsche, editors, *Proceedings of 3rd International United Information Systems Conference, April 21-24, Sydney, Australia*, UNISCON'09, pages 315–327. Springer-Verlag, 2009.

[171] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets. In Andy Schürr and Bran Selic, editors, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, October 4-9, Denver, CO, USA*, MoDELS'09, pages 727–732. Springer-Verlag, 2009.

[172] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Thomas Reiter, Werner Retschitzegger, and Wieland Schwinger. Lets's Play the Token Game – Model Transformations Powered By Transformation Nets. In Daniel Moldt, editor, *Proceedings of the International Workshop on Petri Nets and Software Engineering, in conjunction with 30th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency, June 22-23, Paris, France*, PNSE'09, pages 35–50. Université Paris, 2009.

[173] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging – Second Edition*. Morgan Kaufmann, 2009.



Johannes Schönböck

Education

since 2011 **PhD Studies**, *Technical University Vienna*, Karlsplatz 13, 1040 Wien.
PhD studies in computer science

2008–2011 **PhD Studies**, *Johannes Kepler University Linz*, Altenbergerstrasse 69, 4040 Linz.
PhD studies in computer science

2001–2005 **Diploma Studies**, *Upper Austria University of Applied Sciences*, Softwarepark 11, 4232 Hagenberg, Degree Program Software Engineering for Business and Finance.
Diploma studies in software engineering

1995–2000 **Commercial Academy**, *Handelsakademie Perg*, Dirnbergerstraße 41, 4220 Perg.

Master thesis

Title: *MDA-Editor zur Generierung von Applikationsrahmen für mobile Endgeräte*

Supervisor: Dr. Werner Kurschl

Honours and Awards

Best Thesis Award *FH-Hagenberg 2005: First Place for Diploma Thesis "MDA-Editor zur Generierung von Applikationsrahmen für mobile Endgeräte"*

Experience

since 2009 **Research Associate**, *Institute of Software Technology and Interactive Systems*, Vienna University of Technology, Favoritenstrasse 9-11/188-3 13, 1040 Vienna.

since 2006 **Teaching activities**, *FH OÖ Studienbetriebs GmbH*, Softwarepark 11 4232 Hagenberg.

- Lecture in Model Driven Engineering
- Practical lecture in Programming
- Practical lecture in Mobile Computing

2005–2009 **Research Associate**, *FH OÖ Studienbetriebs GmbH*, Softwarepark 11 4232 Hagenberg.

Projects

since 2009 **FWF-Research Project**, *TROPIC - Transformations on Petri Nets in Color.*

Lina 1 – 4311 Schwertberg
0699/12 14 82 79 • ☎ 07262/585 85 • ✉ schoenboeck@big.tuwien.ac.at
• www.big.tuwien.ac.at

2008–2009 **BF OÖ**, *Telehomecare*.
2007–2008 **KPlus**, *Plant Safety System - Global Tracking of Technicians for Industrial Purposes*.
2005–2007 **FHplus**, *Gulliver - Speech Recognition Services for Wireless Mobile Devices*.

Personal Data

Date of Birth 29 October 1980 in Linz
Place of Birth Linz
National Status Austria
Marital status single

Languages

German **native language**
English **fluent**
French **basic skills**

Community Service

Chairman of music orchestra MV Schwertberg

Publications

[1] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 2011. Accepted for Publication.

[2] Gerti Kappel, Johannes Schönböck, Manuel Wimmer, Gabriele Kotsis, Angelika Kusel, Birgit Pröll, Werner Retschitzegger, Wieland Schwinger, and Stephan Lechner. TheHiddenU - A Social Nexus for Privacy-Assured Personalisation Brokerage. In Joaquim Filipe and José Cordeiro, editors, *Proceedings of the 12th International Conference of Enterprise Information Systems, June 8 - 12, Funchal, Madeira, Portugal*, ICEIS'2010. INSTICC Press, 2010.

[3] Werner Kurschl, Wolfgang Gottesheim, Stefan Mitsch, Rene Prokop, and Johannes Schönböck. Evaluation of a Mobile Multimodal Application Design - Major Usability Criteria and Usability Test Results. In *Proceedings of International Conference on Mobile Business, July 9-11, 2007, Toronto, Ontario, Canada*, ICMB'07, page 68. IEEE Computer Society, 2007.

[4] Werner Kurschl, Wolfgang Gottesheim, Stefan Mitsch, Rene Prokop, and Johannes Schönböck. Automatic Position Determination of Fixed Infrastructure Sensor Network Nodes based on Topology Sensing and Maps. In Hamid R. Arabnia and Victor A. Clincy, editors, *Proceedings of the 2008 International Conference on Wireless Networks, July 14-17, Las Vegas, Nevada, USA*, ICWN'08, pages 17–22. CSREA Press, 2008.

[5] Werner Kurschl, Wolfgang Gottesheim, Stefan Mitsch, Rene Prokop, and Johannes Schönböck. Tele-Homecare - Medical Assistance and Habitability Improvement for Elderly People through Ambient Intelligence. In *Proceedings of FFH 2008, Wels, Austria*, 2008.

[6] Werner Kurschl, Wolfgang Gottesheim, Stefan Mitsch, Rene Prokop, Johannes Schönböck, and Wolfgang Beer. A Two-Layered Deployment Scheme for Wireless Sensor Network based Location Tracking. In *Proceedings of Fifth International Conference on Information Technology:*

New Generations, 7-8 April 2008, Las Vegas, Nevada, USA, ITNG'08, pages 726–730. IEEE Computer Society, 2008.

[7] Werner Kurschl, Wolfgang Gottesheim, Stefan Mitsch, Rene Prokop, Johannes Schönböck, and Wolfgang Beer. Large-Scale Industrial Positioning and Location Tracking - Are We There Yet? In *Proceedings of 7th International Conference on Mobile Business, July 7-8, Barcelona, Spain*, ICMB'08, pages 251–259. IEEE Computer Society, 2008.

[8] Werner Kurschl, Stefan Mitsch, Rene Prokop, and Johannes Schönböck. Model-Driven Development of Speech-Enabled Applications. In *Proceedings of the FH Science Day, Hagenberg, Austria*, pages 216–223, 2006.

[9] Werner Kurschl, Stefan Mitsch, Rene Prokop, and Johannes Schönböck. Development Issues for Speech-Enabled Mobile Applications. In Wolf-Gideon Bleek, Jörg Raasch, and Heinz Züllighoven, editors, *Proceedings of Software Engineering 2007, Fachtagung des GI-Fachbereichs Softwaretechnik, March 27-30, Hamburg, Germany*, SE'08, pages 157–168. GI, 2007.

[10] Werner Kurschl, Stefan Mitsch, Rene Prokop, and Johannes Schönböck. Gulliver-A Framework for Building Smart Speech-Based Applications. In *Proceedings of 40th Hawaii International International Conference on Systems Science, January 3-6, Waikoloa, Big Island, HI, USA*, HICSS-40, page 30. IEEE Computer Society, 2007.

[11] Werner Kurschl, Stefan Mitsch, and Johannes Schönböck. An Engineering Toolbox to Build Situation Aware Ambient Assisted Living Systems. In Johnson I. Agbinya, Elmarie Biermann, Yskandar Hamam, Ntsibane Ntlatlapa, and Keith Ferguson, editors, *Proc of Third International Conference on Broadband Communications, Information Technology & Biomedical Applications, November 23-26, Pretoria, Gauteng, South Africa*, Broadcom'08, pages 110–116. IEEE Computer Society, 2008.

[12] Werner Kurschl, Stefan Mitsch, and Johannes Schönböck. MODEL-DRIVEN PROTOTYPING SUPPORT FOR PERVASIVE HEALTH CARE APPLICATIONS. In T. F. Gonzalez, editor, *Proceedings of IASTED Internation Symposium on Distributed Sensor Networks, November 16-18, Orlando, Florida, USA*, DSN 2008, pages 118–123, 2008.

[13] Werner Kurschl, Stefan Mitsch, and Johannes Schönböck. An Evaluation Framework for Pervasive Healthcare Applications. In *Proceedings of 4th International Conference on Broadband Communications, Information Technology and Biomedical Applications , Wroclaw, Polen*, 2009.

[14] Werner Kurschl, Stefan Mitsch, and Johannes Schönböck. Model-Driven Prototyping Support for Pervasive Healthcare Applications. In Antonio Coronato and Giuseppe De Pietro, editors, *Pervasive and Smart Technologies for Health Care: Ubiquitous Methodologies and Tools*, IGI Global. 2009.

[15] Werner Kurschl, Stefan Mitsch, and Johannes Schönböck. Modeling Distributed Signal Processing Applications. In Benny P. L. Lo and Paul Mitcheson, editors, *Proceedings of Sixth International Workshop on Wearable and Implantable Body Sensor Networks, June 3-5, Berkeley, CA, USA*, BSN'09, pages 103–108. IEEE Computer Society, 2009.

[16] Werner Kurschl, Stefan Mitsch, and Johannes Schönböck. Modeling Situation-Aware Ambient Assisted Living Systems for Eldercare. In *Proceedings of Sixth International Conference on Information Technology: New Generations, April 27-29,Las Vegas, Nevada*, ITNG'09, pages 1214–1219. IEEE Computer Society, 2009.

[17] Werner Kurschl, Stefan Mitsch, Johannes Schönböck, and Wolfgang Beer. Modeling Wireless Sensor Networks Based Context-Aware Emergency Coordination Systems. In Gabriele Kotsis, David Taniar, Eric Pardede, and Ismail Khalil Ibrahim, editors, *Proceedings of 10th International Conference on Information Integration and Web-based Applications Services, 24-26 November 2008, Linz, Austria*, iiWAS'2008, pages 117–122. ACM, 2008.

[18] Werner Kurschl, Stefan Mitsch, Johannes Schönböck, Wolfgang Beer, Wolfgang Gottesheim, and Rene Prokop. Towards a Unified Location Tracking System for Heterogeneous Industrial Environments. In *Proceedings of 6th IEEE International Conference on Industrial Informatics, July 13 - 16, Daejeon, Korea*, number INDIN'08, pages 1267–1272, 2008.

[19] Johannes Schönböck. Modellbasierte MIDP-Entwicklung mit MDA4ME. *Java Spectrum*, January 2006.

[20] Johannes Schönböck. *MDA4ME: Ein MDA-Editor zur Generierung von Applikationsrahmen für mobile Endgeräte*. Vdm Verlag Dr. Müller, 2008.

[21] Johannes Schönböck. Transformation Nets: A Runtime Model for Transformation Languages. In *Proceedings of the Doctoral Symposium at Model Driven Engineering Languages and Systems, October 4-9, Denver, CO, USA*, MoDELS'09. School of Computing, Queen's University, 2009.

[22] Johannes Schönböck, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Catch Me If You Can - Debugging Support for Model Transformations. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009. Reports and Revised Selected Papers*, pages 5–20. Springer, LNCS 6002, 2010.

[23] Johannes Schönböck, Florian König, Gabriele Kotsis, Dominik Gruber, Emre Zaim, and Albrecht Schmidt. MirrorBoard - An Interactive Billboard. In Michael Herczeg and Martin Christof Kindsmüller, editors, *Proc. of Mensch & Computer 2008: Viel Mehr Interaktion, Interdisziplinäre Fachtagung, 7.-10. September 2008, Lübeck, Germany*, pages 217–226. Oldenbourg Verlag, 2008.

[24] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Right or Wrong? - Verification of Model Transformations using Colored Petri Nets. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling in conjunction with 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 25 - 29, Orlando, Florida, USA*, DSM'09. Helsinki Business School, 2009.

[25] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Plug & Play Model Transformations - A DSL for Resolving Structural Metamodel Heterogeneities. In *Proceedings of the 10th Workshop on Domain-Specific Modeling in conjunction with Systems, Programming, Languages and Applications Software for Humanity, October 17-21, 2010, Reno, Nevada, USA*, DSM'10. Online Publication, 2010.

[26] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In Laurence Tratt and Martin Gogolla, editors, *Proceedings of the 3rd International Conference on Model Transformation, June 28 - July 2, Malaga, Spain*, ICMT'10, pages 260–275. Springer-Verlag, 2010.

[27] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Taming the Shrew - Resolving Structural Heterogeneities with Hierarchical CPN. In Daniel Moldt, editor, *Proceedings of the International Workshop on Petri Nets and Software Engineering in conjunction with 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency*, PNSE'10, pages 141–157. University of Hamburg, 2010.

[28] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings of the First International Workshop on Model-Driven Interoperability (MDI 2010) in conjunction with 13th International Conference on Model Driven Engineering Languages and Systems, October 3-8, Oslo, Norway*, MoDELS'10, pages 32–41. ACM Press, 2010.

[29] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. From the Heterogeneity Jungle to Systematic Benchmarking. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, volume 6627, pages 150–164. Springer Lecture Notes in Computer Science, 2011.

[30] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris Kolovos, Richard Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In Jordi Cabot and Eelco Visser, editors, *Proceedings of the 4th International Conference on Model Transformations, June 27-28, Zurich, Switzerland*, ICMT'11, pages 31–46. Springer-Verlag, 2011.

[31] Manuel Wimmer, Gerti Kappel, Johannes Schönböck, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. A Petri Net based Debugging Environment for QVT Relations. In *Proceedings of the 24th International Conference on Automated Software Engineering, November 16-20, Auckland, New Zealand*, ASE'09, pages 1–12. IEEE Computer Society, 2009.

[32] Manuel Wimmer, Gerti Kappel, Johannes Schönböck, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. TROPIC: A Framework for Model Transformations on Petri Nets in Color. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 25 - 29, Orlando, Florida, USA*, OOPSLA 2009, pages 783–784. ACM, 2009.

[33] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets. In Andy Schürr and Bran Selic, editors, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, October 4-9, Denver, Colorado, USA*, MoDELS'09, pages 727–732. Springer-Verlag, 2009.

[34] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Thomas Reiter, Werner Retschitzegger, and Wieland Schwinger. Lets's Play the Token Game – Model Transformations Powered By Transformation Nets. In *Proceedings of the International Workshop on Petri Nets and Software Engineering in conjunction with 30th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency, June 22-23, Paris, France*, PNSE'09, pages 35–50. Université Paris 13, 2009.

[35] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. On using Inplace Transformations for Model Co-evolution. In *Proceedings of the 2nd International Workshop on Model Transformation with ATL in conjunction with 3rd International Conference on Model Transformation, June 28 - July 2, Malaga, Spain*, MtATL'10. INRIA & Ecole des Mines de Nantes, 2010.

Lina 1 – 4311 Schwertberg
0699/12 14 82 79 • ☎ 07262/585 85 • ✉ schoenboeck@big.tuwien.ac.at
• www.big.tuwien.ac.at