Design and Analysis Techniques for Precise Model Transformations in Model-Driven Development

Dissertation for the Doctoral Degree of the Hungarian Academy of Sciences

Dániel Varró

Budapest, 2011

dc_244_11

dc_244_11

PRECÍZ MODELLTRANSZFORMÁCIÓK TERVEZÉSE ÉS ANALÍZISE A MODELLVEZÉRELT FEJLESZTÉSBEN

MTA DOKTORI ÉRTEKEZÉS

VARRÓ DÁNIEL

BUDAPEST, 2011

dc_244_11

SUMMARY

Model-driven design (MDD) of complex IT systems has become a popular software engineering paradigm in the last decade, especially for the development of critical dependable systems where formal mathematical proofs are necessitated to assure that the system under design is free of conceptual flaws. Model transformation is a key technology in MDD for the automated bidirectional synchronization of high-level system models and low-level formal models as well as for code generation starting from engineering models of proven quality.

This dissertation presents novel methods and techniques for mathematically precise yet intuitive specification, design, implementation and formal analysis of model transformations aligned with the best practices of model driven software engineering. (1) Based upon an innovative combination of graph transformation and abstract state machines, I define a generic model transformation language to specify model transformations within and between modeling languages. (2) I introduce the "model transformation by example" approach for the semi-automated synthesis of model transformation rules. (3) I proposed novel execution principles, strategies and architectures for model transformations such as model-specific search plans, incremental transformations and compiled transformation plugins. (4) Finally, I elaborate a formal termination analysis technique for model transformations captured by means of graph transformation rules, which is based on the algebraic analysis of a Petri net abstraction.

These techniques enable the systematic design of a wide range of model transformations for model driven engineering of critical dependable systems by automating early model-based analysis and subsequent code generation. The exploitation of the results was primarily carried out in the scope of the VIATRA2 model transformation framework.

ÖSSZEFOGLALÁS

A modellvezérelt tervezés napjainkra egy széleskörűen elterjedt paradigma különösen kritikus beágyazott és szolgáltatás-orientált szoftverrendszerek fejlesztésékor, ahol formális matematikai bizonyítások felhasználásával kell garantálnunk a tervezés alatt álló rendszer hibamentes voltát. Kutatásaim középpontjában az automatikus modelltranszformációk álltak, amelyek többek között a modellvezérelt tervezés során előállított magas szintű rendszermodellek és a verifikációs és validáció során használt formális modellek közötti kétirányú átalakításokat végzik.

Disszertációmban új módszereket dolgoztam ki a modelltranszformációk matematikailag precíz, intuitív és a modellvezérelt szoftvertervezés létező mérnöki gyakorlatához illeszkedő specifikációjára, tervezésére, végrehajtására és formális analízisére. (1) Kidolgoztam egy, a gráftranszformáció és absztrakt állapotgépek újszerű kombinációjára épülő generikus modelltranszformációs nyelvet a modellezési nyelveken belüli és modellezési nyelvek közötti transzformációk definiálására. (2) Elsőként definiáltam a modelltranszformáció-példák-alapján megközelítést a modelltranszformációs szabályok fél-automatikus származtatására. (3) Hatékony modelltranszformációs végrehajtási stratégiákat, architektúrákat és elveket javasoltam (modellspecifikus keresési tervek, inkrementális modelltranszformációk és lefordított transzformációs programok). (4) Továbbá kidolgoztam egy Petri hálós absztrakción alapuló formális analízis módszert a gráftranszformációs szabályokkal specifikált modelltranszformációk terminálásának vizsgálatára.

E technikák lehetővé teszik a modelltranszformációk egy széles körének szisztematikus tervezhetőségét a kritikus rendszerek modellvezérelt tervezése területén, automatizálva elsődlegesen a korai formális modellanalízis és a kódgenerálás lépéseit. E kutatási eredmények gyakorlati hasznosulását a VIATRA2 modelltranszformációs szoftverrendszer (mint mérnöki alkotás) tette lehetővé, amelynek alapítója és kutatási vezetője vagyok.

ACKNOWLEDGEMENTS

My research results could not have been achieved without the support of numerous people to whom I am deeply indebted.

First, I am grateful to Prof. András Pataricza introducing me the topic of model transformations back in 1999, then guiding me through the troubled waters of my PhD studies (from 2000 to 2004), and integrating me to the Fault Tolerant Systems Research Group by continuously offering new scientific and non-scientific challenges.

Right next comes the core VIATRA team, which consists of adventurous young researchers and PhD students including István Ráth, Ákos Horváth, Gábor Bergmann, Ábel Hegedüs and Zoltán Ujhelyi, who dared to select a young and inexperienced supervisor (like me) to guide their scientific activities. Throughout the VIATRA project, they really acted as a team helping each other at numerous occasions - also reducing my own supervisory duties. An important role is played by those colleagues (namely, András Balogh, László Gönczy) where I acted as a co-tutor of their PhD studies. Furthermore, I would like to say thank you to several colleagues (including István Majzik, Gergely Pintér, Zoltán Micskei, Balázs Polgár , Dániel Tóth and many more) at the Department of Measurement and Information Systems whom I actively collaborated in different research projects.

I would like to express my grateful attitude to numerous highly respected international collaborators who provided scientific as well as moral support during the years. I would like to name Hartmut Ehrig (Berlin), Gregor Engels (Paderborn), Reiko Heckel (Paderborn / Leicester), Arend Rensink (Univ. Twente), Juan de Lara (Madrid), John Rushby (SRI International), Andy Schürr (Darmstadt) and Gabi Taentzer (Berlin / Marburg) and many of their close colleagues. This list should transitively include all collaborators in joint European research projects, especially SENSORIA (led by Martin Wirsing), DIANA (by Tobias Schoofs), SecureChange (run by Fabio Massacci) which resulted in some really fruitful long-term collaborations.

Last but not least, in my case, a really unique contribution comes from various members of my family. I started to work on model transformations with my brother, Gergely Varró in 1999 which lasts until today. The Family Research Corporation also includes my wife, Szilvia Varró-Gyapay with whom I spent memorable research visits in Berlin in 2004 and 2005 working together in the field of graph transformation. Her support is also invaluable in running our family and taking care of our boys, Balázs and Csaba while I was writing some more “silly papers” - just like that of my parents, Győző and Mária, who were always available as a substitution whenever needed.

Contents

dc_244_11

Chapter 1

Introduction

1.1 An Introduction to Critical Systems and Services Design

The critical systems and services design is still a major challenge in software engineering. Whole enterprises depend on business-critical services, and a malfunction of such services frequently results in heavy financial losses. Moreover, these services need to be robust and adaptive to frequent changes in the business environment. In case of safety-critical systems, the failure of a critical component may result in severe damages or even casualties. As a consequence, one has to demonstrate that the system under design is free of design and implementation flaws by rigorous verification and validation techniques during a certification process.

1.1.1 Certification artifacts in critical systems design

During a certification process, various software-related **certification artifacts** are designed and validated. Here, we provide a brief overview of the main certification artifacts by extracting relevant parts from related standards and industrial recommendations [97, 120].

High-Level Requirements (HLR) are software requirements developed from analysis of system requirements, safety-related requirements, and system architecture [120]. This definition implies a black-box view of the software, i.e., the high-level requirements are created primarily from system considerations and are not dependent on and do not define the software architecture. HLRs are still most frequently captured textually in a natural language [97].

Derived Requirements are additional requirements resulting from the software development process, which may not be directly traceable to higher level requirements [120]. Derived requirements are introduced during the software development process as the result of design decisions. They modify or further constrain the externally visible behavior of the system or they have safety implications [97].

Low-Level Requirements (LLR) are software requirements derived from high-level requirements, derived requirements, and design constraints from which source code can be directly implemented without further information [120]. Low-level requirements can also be specified in a model that specify structure and behavior on a high-level of abstraction [97].

Software Architecture (SA) refers to the structure of the software selected to implement the software requirements [120]. The software architecture defines what software components are to exist, the interfaces to those components, how components are scheduled and invoked, and how information flows between the components [97]. Partitioning of data and control is also a function of the software architecture. The software architecture typically corresponds to models being specific to the execution platform, which, for instance, already incorporate design decisions for optimization steps.

Source Code (SC) is the code written in source languages, such as assembly language and/or high level language, in a machine readable form for input to an assembler or compiler [120]. Currently, most development environments produce traditional source code such as C or Ada as part of the translation process [97].

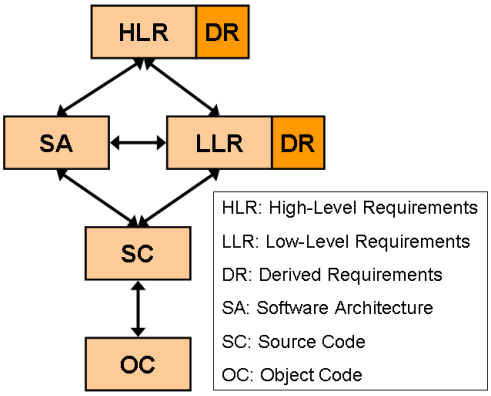


Figure 1.1: Certification artifacts

The final **Executable Object Code (EOC)** is obtained by traditional compilers, which compilation step is outside the scope of the current thesis, and the Executable Object Code can be loaded into the target hardware and executed without further problems.

The certification process requires that all outputs of the software requirements, design and coding process are verifiable, conformant to standards, traceable, accurate and consistent, and compliant with artifacts on higher-levels of abstraction. More specifically, (1) LLRs should be consistent with HLRs, (2) the selected software architecture (SA) should be in conformance with HLRs, (3) the software architecture is aligned with LLRs, (4) source code (SC) complies with both the LLRs and software architecture (SA). An overview of these verification and validation tasks between certification artifacts is presented in Fig. 1.1.

1.1.2 Verification and validation techniques in certification

In most of the cases, compliance is assured by using testing and/or formal methods.

Testing. Traditionally, verifying the compliance of the implementation with respect to requirements (HLR and LLR) are carried out by **testing**. The certification process should guarantee that (i) test cases exist for each software requirement, (ii) appropriate test coverage is achieved by these test cases, and (iii) regular tests are complemented with robustness tests to investigate abnormal inputs and conditions [97, 120].

Test procedures are still frequently created manually and this objective is satisfied by a review of the test procedures to make sure they were correctly developed. In certain cases, it may be possible to automatically generate test cases and test procedures from the requirements [97]. Since the requirements-based test cases may not have completely exercised the code structure, structural coverage analysis (and potentially additional testing) is performed to provide structural coverage in accordance with the required software criticality level.

Formal methods. As the main alternative, verification and validation activities in a certification process are carried out by using **formal methods** (such as Petri nets, transition systems, process algebra, etc.). In such a case, a formal proof is constructed that the software under design is free of flaws. Primary means of verification and validation include theorem proving, model checking or static analysis techniques. **Model checking** automatically investigates the validity of a requirement by systematically traversing all possible trajectories (execution paths) of a system model. In case of **theorem proving**, properties are aimed to be proved for all possible models by semi-automated deductive reasoning techniques. Finally, **static analysis** directly investigates the source code (and the operational semantics of the program or the model) to reveal inconsistencies without the execution of the code.

1.2 Model Transformations in Model Driven Development

1.2.1 Model-driven development in services and systems engineering

Model-driven development (MDD) (and closely related concepts, like Model Driven Architecture – MDA [94] or Model Integrated Computing – MIC [132]) has recently become a key technique in critical software and systems engineering. MDD facilitates the extensive and systematic use of models from the very early phase of the design cycle, as illustrated in Fig. 1.2. System requirements and design are captured by high-level, visual **engineering models** (using popular and standardized modeling languages like UML [106], SysML [109], AADL [121]). Early systematic **formal analysis of design models** can be carried out by generating appropriate mathematical models by **automated model transformations (MT)**. Formal analysis retrieves a list of problems, which can be **back-annotated** to the high-level engineering models to allow system designer to make corrections prior to investing in manual coding for implementation. This way, formal methods are hidden by automated model transformations which project system models into various mathematical domains [24, 133]. High-level system models can also be a basis of additional optimization steps [76, 92]. Finally, the source code of the target system is derived by **automatic code generation** from the provenly correct and optimized system model. Moreover, additional model transformations may also automate the generation of runtime monitors and deployment descriptors for the target reliable platforms (like AUTOSAR [15] for the automotive or ARINC653 [7] for the avionics domain) or control the system under operation [127].

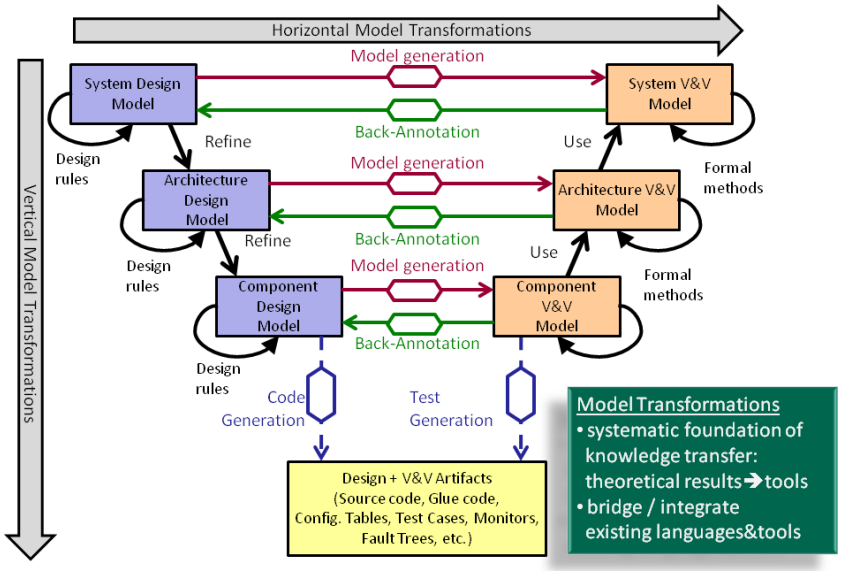


Figure 1.2: Model-driven development for critical systems

1.2.2 Advantages of model-driven development

Recent reports demonstrated significant increase both in productivity and quality using a model-driven development with automated code generators (like SCADE in the avionics domain or IBM Websphere Business Modeler used for business process modeling). For instance, development costs of avionics systems were reduced by 50%, while testing costs were reduced by

10-30% [98, 111] thanks to the automatic generation of safe, production quality source code compliant with DO-178B Level A certification standard [120]. Altogether, the total certification costs have been reduced by more than a factor of two by partly eliminating manual coding, code reviews, and providing unquestionable traceability between requirements and various design artifacts. Similar figures have been reported in the service-oriented domain by using a combination of precise business process modeling and formal analysis [56].

Automated model transformations can provide an efficient method to integrate and transfer theoretical results of formal model analysis, code and configuration generation into practice [29, 56, 122]. Such transformations have also been intensively investigated in various European research projects (like TOPCASED, DECOS, ASSERT, or DIANA in the safety-critical domain, and DEGAS, DEPLOY or SENSORIA in the service-oriented domain).

1.2.3 Overview of Model Transformations

Conceptually, all translations from one (or more) source modeling language to one (or more) target language are commonly regarded as **model transformations** [40], which are thus key factors in a successful adoption of MDD [64]. The main concepts of model transformations are summarized in Fig. 1.3.

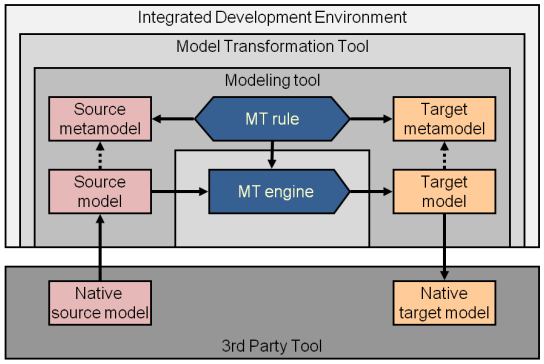


Figure 1.3: Model transformation concepts

First the source and target language of the transformation needs to be defined by their respective **metamodels**. At design time, a model transformation is defined by a set of **transformation rules**. At execution time, in a **model transformation run**, transformation rules are executed by a **model transformation engine** to generate a **target model** (as output) from a given **source model** (as input). Since **native source and target models** of third party tools are frequently provided as some textual files, source models are frequently obtained by using **model importers**, while target models are post-processed by **model exporters**.

1.3 Research Challenges and Objectives

Several industrial and academic usage scenarios have demonstrated that a precise software engineering approach is necessitated to cover the entire life-cycle for model transformation development including their specification, design, execution and validation.

1.3.1 Specification of Model Transformations?

Despite the availability of various model transformation tools, in the de facto industrial practice, many model transformations are still written manually in as a regular piece of software in a rather ad hoc way. This is partly due to the fact that the specification languages of existing model transformations are either precise but not intuitive (which prevent widespread industrial application), or intuitive but not precise (which is necessitated in the critical systems and services domain). Furthermore, unlike popular development environments, model transformation frameworks rarely offer a library of generic and reusable transformation components.

The evolution trend of model transformation languages is characterized by gradually increasing the abstraction level of such languages to declarative, rule-based formalisms as promoted by the QVT (Queries, Views and Transformations) [108] standard, which unfortunately, lacks a precise formal underpinning, leading into various, incompatible execution semantics [67]. Now the first challenge of the thesis is formulated as follows.

Challenge 1 (Specification of Model Transformations) *Model transformations require a mathematically precise yet intuitive specification language, which offers to construct generic, reusable model transformation components.*

1.3.2 Design of Model Transformations?

The efficient design of automated model transformations between modeling languages has become a major challenge to model-driven development. Many highly expressive transformation languages have emerged to provide support (like ATL [77], ATOM3 [42], AGG [54], EPSILON [118], FUJABA [103], GReAT [16], GrGEN.NET [63], Henshin [10], MOFLON [9], MOLA [81], Tefkat [87], VIATRA2 [J14, K9], VMTS [90]).

However, a common deficiency of all these languages is that their transformation language is substantially different from the source and target models they transform. As a consequence, transformation designers need to understand not only the transformation problem, i.e. how to map source models to target models, but significant expertise is required in the transformation language itself to formalize the solution. Unfortunately, many domain experts, who are specialized in the source and target languages, lack such skills in underlying transformation technologies. As a consequence, the second challenge of the current thesis is the following.

Challenge 2 (Design of Model Transformations) *Model transformations require powerful design techniques, which enable domain experts without expertise in the underlying transformation technology to develop model transformations by themselves.*

1.3.3 Execution of Model Transformations?

In complex industrial scenarios, model transformations need to query and manipulate models with hundred thousands or millions of model elements. Well-known examples are AUTOSAR models [15] in the automotive domain, and various SysML models in avionics. Unfortunately, most model transformation tools fail to handle models of extreme size, which significantly limit their applicability. Existing transformation tools of the Eclipse M2M framework failed to demonstrate their scalability for handling models over 100 000 model elements.

In addition to this performance issue, model transformations rules can typically be executed within a complex transformation framework, which prevents to embed the actual transformations themselves in third party tools. For instance, if a model transformation aims at providing a bridge between two database schemas, then it would be advantageous to embed the model transformations to database technology, e.g. in the form of SQL scripts, or native Java programs. In other terms, individual model transformations should be integrated easily into designated third-party (back-end) tools without integrating the model transformation tools them.

Due to these issues, model transformations tools in the Eclipse-based development tools are limited to carry out small to medium scale transformations despite the conceptual clarity of the transformation approach. Therefore, the third challenge can be formulated as follows:

Challenge 3 *Model transformations require efficient execution techniques, which handle models with millions of elements, and enable the low-cost integration of model transformations in popular development frameworks and third-party tools.*

1.3.4 Correctness of Model Transformations?

Unfortunately, even automated model transformations can be erroneous, which would invalidate the results of a thorough mathematical analysis [82, K24]. Therefore, one has to guarantee the model transformations themselves are free of design errors, thus no design flaws have been introduced to the target model during the model transformation process. Otherwise, when some problems are reported during mathematical analysis, it is impossible to distinguish whether it is due to erroneous system design or a flaw in the model transformation.

Existing research results for the verification and validation of model transformations are in a very early stage: they address ad hoc challenges, and no scalable techniques are available for industrial use. Moreover, existing validation techniques of model transformations are incompatible with the artifacts required by a certification process of critical systems. As a result, all target models generated by a transformation has to be treated as if it were created manually by a designer, and a time consuming revalidation is necessitated.

Challenge 4 *Model transformation necessitate mathematically well-founded verification and validation techniques to provide formal assurance for their correctness.*

1.3.5 Objectives

In this thesis, I elaborate novel techniques for the **precise specification, design, execution and formal analysis of model transformations**. These techniques enable the systematic engineering of a wide range of model transformations in a model-driven design of critical systems to automate early model-based analysis as well as code generation.

1.4 Towards Novel Scientific Results

Now I briefly summarize the research method related to addressing each scientific challenge.

1.4.1 Specification Techniques for Model Transformations

Research Method. Critical systems frequently require to carry out a thorough certification process to justify that the system under design meets its (formal) specification. In critical systems, the semantics of the domain-specific languages used during various phases of the development process is precisely captured by formal specification techniques to avoid incomplete and ambiguous specification of the system under design. Since critical systems serve as a main application domain for model transformations, the same level of scrutiny is thus necessitated for the specification of model transformations themselves.

Nowadays, model transformations are typically designed by software engineers having expert knowledge in both the source and target languages of the transformation. However, unlike in case of the majority of programming languages, most model transformations are data driven. As a consequence, a declarative transformation language would be advantageous. On the other hand, software engineers are more familiar with conventional (procedural or object-oriented) programming languages, which hinders the easy adoption of a fully declarative model transformation language.

As a result, the *specification of model transformations* in a critical systems context *necessitates the use of mathematically precise, yet intuitive formalisms*.

In order to address genericity and reusability, two major ideas are investigated. First, generic transformations like transitive closure are independent of the application domain (i.e. source and target languages of the transformation), thus the *specification of transformations may allow*

the use of type parameters or type variables. This is in close correspondence with the generics (templates) of object-oriented programming languages (like Java or C++).

As a consequence, *model transformation formalisms should be general* enough to provide reusable transformation components or libraries, which can be easily tailored to various transformation problems.

After investigating the use of two popular formal specification techniques, namely, graph transformation [51] and abstract state machines [70], I found that (i) graph transformation rules offer an intuitive and declarative way for capturing elementary transformation rules, but complex transformation programs are difficult to be assembled in a purely declarative paradigm. Moreover, (ii) abstract state machines can straightforwardly describe arbitrary algorithms [70], but the algebraic representation of complex (graph-like) model structures becomes complicated.

Novel Results. As Contribution 1 (in Chapter 3), I propose an innovative adaptation of graph transformation and abstract state machines as a hybrid and generic model transformation language to combine the advantages of a declarative and a procedural paradigm also supporting to assemble reusable model transformation libraries.

Validation of Results. As a practical validation of the language, we developed the VIATRA2 model transformation framework [J14], in order to support model transformations applied in various phases of MDD investigated in numerous research and industrial projects. VIATRA2 as a software system is also one of the major engineering result related to the current thesis.

1.4.2 Design Techniques for Model Transformations

Research Method. In order to assist domain experts in creating model transformations, I investigated alternate ways for synthesizing rule-based knowledge representations. Interestingly, several "by-example" approaches have been proposed in different fields of software engineering to synthesize knowledge by interacting with domain experts who supply prototypical examples as primary input.

In advanced XML transformer tools, XSLT rules are generated automatically after relating simple source and target XML documents [55, 88, 110, 140]. Query-by-example [141] aims at proposing a language for querying relational data constructed from sample tables filled with example rows and constraints. Programming by example [39, 116], where the programmer (often the end-user) demonstrates actions on example data, and the computer records and possibly generalizes these actions, has also proven quite successful.

When taking a by-example approach in the context of model transformations, the primary goal is to *automate the derivation of model transformation rules*. However, as existing results of by-example in other research fields indicated, full automation is rarely possible, thus an iterative interaction with the transformation designers will be necessitated.

As the formal specification of models and transformations can be easily represented in a logic programming language, my attention turned to various logic programming approaches, especially, inductive logic programming (ILP) [101]. The latter provides a mathematically well-founded way for infer rules of Prolog programs as a hypothesis derived from a knowledge base and a set of positive and negative examples.

Novel Results. As Contribution 2 (in Chapter 4), I propose a novel way of automating the construction of transformation rules taking a *by-example approach* in the context of model-driven development.

Validation of Results. The practical validation of the results were carried out by developing a prototype tool (another major engineering result of the current thesis). Using the tool, we carried out three complex model transformation problems taken from various application scenarios.

1.4.3 Efficient Execution Techniques for Model Transformations

Research Method. In practical applications of model transformations, the size of source and target models frequently exceeds one million model elements. In order to manage this complexity, well-founded, efficient techniques are required to query and manipulate the underlying models.

Existing model transformation frameworks (in the beginning of the current line of research, back in 2004) (1) offered a *batch-oriented approach for model transformations*, when source models are transformed into their target equivalents upon explicit user request. However, such a batch model transformation approach is ill-suited for various industrial scenarios like incremental synchronization between various models for tool integration [85], model migration [118] or rule-based model simulation of discrete event-based systems.

This way, I elaborated the concepts of incremental model transformations based upon the paradigm of graph transformation. In case of incremental model transformations, one main problem is to identify which parts of a transformation need too be re-executed in response to changes of the source model. Since a specific model transformation rule is typically matched by a relatively low number of source model elements (compared to the size of the models themselves), the explicit storage of transformation rule matches seemed to be viable approach.

Moreover, in existing transformation frameworks, (2) the execution of model transformation rules only *incorporated metamodel-level knowledge for navigating and querying models* in the pattern matching phase. However, a model transformation rule can be executed in different ways according to different execution strategies (aka search plans), and the actual structure of the model under transformation may have a significant impact on the effectiveness of a specific search plan. For instance, a search plan needs to make an important decision on where to start a query, e.g. to obtain an initial set of model elements according their type or the containment hierarchy. This way, I focused my investigations how (cheaply maintained) model-specific information can improve the execution strategy of local-search based transformation rules.

Finally, (3) model transformation tools only provided a closed technology, thus *the integration of actual model transformations into off-the-shelf frameworks was problematic*. Furthermore, this causes severe problems in the certification of model transformations and the qualification of model transformation tools as well. Therefore, I investigated how the architecture of a model transformation framework can be modified in order to separate the design and execution phases of model transformations.

Novel Results. As Contribution 3 (in Chapter 5), I elaborate a novel execution architecture and various efficient execution strategies for model transformations.

Validation of Results. In case of the novel execution strategies, numerous benchmark measurements have been carried out (e.g. in [J2, J18, J19, J21, K3, K4, K6, K28]) to validate our conceptual results by experimental evaluation. In fact, we were first to propose benchmark problems [K28] for performance evaluation of graph-based transformation tools. The novel execution architecture was validated by respective prototype implementations for two platforms.

1.4.4 Termination Analysis for Model Transformations

Context. Various correctness criteria have been defined for model transformations [K24]. The minimal requirement is to assure *syntactic correctness*, i.e., to guarantee that the generated

model is a syntactically well–formed instance of the target language. An additional requirement called *syntactic completeness* is to completely cover the source language by transformation rules, i.e., to prove that there exists a corresponding element in the target model for each construct in the source language.

However, in order to assure a higher quality of model transformations, at least the following *semantic requirements* should also be addressed.

- **Termination:** An essential criterion that needs to be guaranteed is a model transformation will terminate. This is a very general, and modeling language independent semantic criterion for model transformations.

- **Uniqueness (Determinism, confluence, functionality):** As non-determinism is frequently used in the specification of model transformations (as in the case of graph transformation based approaches) we must also guarantee that the transformation yields a deterministic result. Again, this is a language independent criterion.

- **Semantic correctness (Property preservation):** In theory, a straightforward correctness criterion would require to prove the semantic equivalence of source and target models. However, as model transformations may also define a *projection* from the source language to the target language (with deliberate loss of information), semantic equivalence between models cannot always be proved. Instead, we define *correctness properties* (which are typically transformation specific) *that should be preserved by the transformation*.

  The syntactic and semantic correctness of transformations are investigated in many papers [13, 89, B5, J12, J13, K21, K24]. The current thesis exclusively focuses on the termination problem of model transformations.

Research Method. The formal analysis of model transformations captured by graph transformation rules is a complicated task, as it allows to query and manipulate complex, graph-like models. In order to come up with efficient analysis techniques powerful abstractions need to be defined. Petri nets are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools (including static analyzers and model checkers). As a result, Petri nets served as a prime candidate as an abstract formal analysis domain for termination analysis of model transformation rules.

Such a Petri net abstraction needs to be correct in the sense that each run of the original graph transformation system needs to have a corresponding run in the Petri net. On contrary, a run of the Petri net may correspond to multiple execution traces in the original graph transformation system, thus an abstraction is necessarily not complete. In such a case, the Petri net simulates the graph transformation system.

My method was to reuse well known results and analysis techniques (e.g. invariants) of Petri net theory for the correctness analysis model transformations. Due to the information loss during abstraction, such an approach will provide a sufficient termination criterion, i.e. it may prove that the original graph transformation system is terminating, or provide a “do not know” (uncertain) answer otherwise.

Novel Results. As Contribution 4 (in Chapter 6), I provide a sufficient termination criterion for model transformations by defining a powerful abstraction of graph transformation rules to Petri nets.

Validation of Results. As this result is primarily of theoretical nature, a formal proof of the related theorems is carried out.

1.5 STRUCTURE OF THE THESIS

The rest of the thesis is structured as follows:

- Chapter 2 provides foundations for modeling languages (based upon metamodeling) and model transformations (by graph transformation). In addition, this chapter also overview some formal methods (namely, Petri nets and inductive logic programming) used in subsequent chapters.
- Chapter 3 introduces a specification language for model transformations obtained by an innovative combination of two formal modeling paradigms. Since 2005, this language serves as the transformation language of the VIATRA2 model transformation framework.
- Chapter 4 presents a novel approach, called model transformation by example, which specifies model transformations by pairs of prototype models, and then derives transformation rules semi-automatically using inductive logic programming.
- Chapter 5 discusses high-level strategies for the efficient execution of model transformations, such as model-specific search plans, incremental model transformations, and standalone model transformation plugins.
- Chapter 6 investigates the termination problem for model transformations formally captured by means of graph transformation systems, and introduces an analysis technique based on a Petri net abstraction.
- Finally, Chapter 7 summarizes the novel contributions of the thesis, and provides a brief outlook to various applications and utilizations of the conceptual contributions.
- As an appendix to complement the main results of the thesis, a case study of the VIATRA2 transformation language and recent benchmark evaluations are presented.

Related work is assessed individually for the four major conceptual chapters. Note, however, that related work is assessed in full depth up to the publication date of the respective results (and not as of today). This way, the main added value of my results can be better assessed. However, some follow-up results in the specific areas will also be briefly presented at various places.

Since this thesis is a composure of results in the field of software engineering and formal methods, this duality is reflected in the style of the thesis to follow the style of academic papers in the respective fields. Therefore, a more formal style is used for discussing semantics and correctness issues, while a more informal style is used for presenting software engineering related aspects (like e.g. languages, execution strategies).

Chapter 2

Preliminaries

This chapter provides an overview on some main concepts of models and model transformations as well as the formal underpinning of various techniques used throughout this thesis. We first introduce domain-specific modeling languages and metamodeling (Sec. 2.1.1) followed by a summary on model transformation specified by graph transformation techniques (Sec. 2.2). Then we overview the concepts of inductive logic programming (Sec. 2.3), which serves as the foundation of model transformation by example to be presented in Chapter 4. Finally, we provide an introduction to Petri nets (Sec. 2.4), which is intensively used later in Chapter 6.

2.1 Domain Specific Modeling Languages and Metamodeling

Metamodeling is a fundamental part of model transformation design as it allows the structural definition (i.e. abstract syntax) of the source and target modeling languages of the transformations. Then a model transformation is operating on models, which are instances of the metamodel. Taking an analogy with traditional formal languages, a metamodel may correspond to the grammar of a formal language, while a model is a sentence of the language.

2.1.1 Domain-specific modeling languages

Aspects of a domain-specific modeling language. From a language engineering perspective, a **domain-specific modeling language (DSML)** consists of the precise definition of

- the **abstract syntax** of the language metamodel, i.e. the core concepts in the language and their relations captured by a metamodel;
- the **concrete syntax** of the language of the language to describe the graphical or textual appearance of the language constructs;
- the **well-formedness constraints** (or static semantics) of the language to capture further restrictions and constraints on valid model instances of the language;
- the **behavioral semantics** (or dynamic semantics) of the language to specify the valid steps or evolution paths for the models of the language; and
- **mappings** to other languages

The concepts of DSMLs incorporate traditional high-level modeling languages used by software, systems or service engineers (like UML, SysML, BPMN, AADL, etc.) as well as popular

formal modeling notations (like Petri nets, transition systems, process algebra, etc.) or programming languages (with existing metamodels for languages like C++, Java and many more).

It is worth pointing out that from a DSML viewpoint, the abstract syntax of the language is the core design artifact. This abstract syntax can be complemented by multiple concrete syntax representations (which may potentially include an alternate graphical and textual representation of the same language like in case of AADL [121]).

Metamodeling frameworks. Metamodels are represented in a **metamodeling language**, which is another modeling language for capturing metamodels. Currently, most widely used metamodeling languages (e.g. Eclipse Modeling Framework (EMF) [48]) are derived with slight variations from the Meta Object Facility (MOF) [105] metamodeling standard issued by the OMG. However, as stated in [J15], the MOF standard fails to support multi-level metamodeling [14], which is typically a critical aspect for integrating different technological spaces [23] where different metamodeling paradigms are used. There are various initiatives like KM3 [78], or VPM (Visual and Precise Metamodeling) [J15] to provide a uniform representation of models, metamodels and transformations. VPM (used in the VIATRA2 model transformation framework) introduces the concept of a *model space* using explicit and generalized instance-of relations. Most contributions in the current thesis are general in the sense that they are independent of the underlying metamodeling framework.

2.1.2 Metamodels of dynamic modeling languages

Obviously, a DSML does not necessarily contain all previous five aspects. For instance, UML class diagrams are a static DSML (without dynamic semantics), while statecharts or Petri nets provide an example for a dynamic modeling language. In case of a **dynamic modeling language (DML)**, various metamodels are usually defined, which are exemplified together with main relationships in Fig. 2.1.

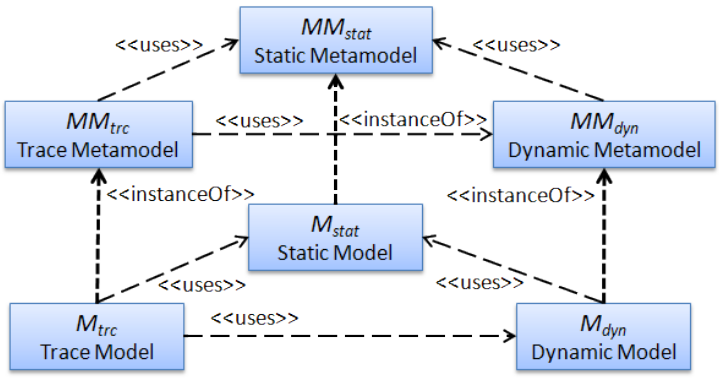


Figure 2.1: Metamodels and instances of a dynamic model

First, a **static metamodel** MM_{stat} defines the static structure of a language including possible types of model elements, their main attributes and relations with other model elements. An instance of this metamodel is called the **static model** (M_{stat}), e.g. a concrete Petri net structure.

Next, a **dynamic metamodel** MM_{dyn} uses and extends the static metamodel MM_{stat} for storing information related to dynamic behavior (e.g. current state, value, configuration) of a structural element. The **dynamic model** (M_{dyn}) is an instance of the MM_{dyn}, e.g. the current marking of a given Petri net place.

Finally, an **execution trace metamodel** (MM_{trc}) is defined for the language to represent runs of the M_{dyn}. The MM_{trc} *uses* the MM_{dyn} for recording how the dynamic model changed

and the MM_{stat} for describing which static element is concerned. An **execution trace model** (M_{trc}) is an instance of the MM_{trc}, e.g. the sequence of fired transitions of a Petri net. The M_{trc} describes the changes of M_{dyn}, therefore it is represented as a **change model** in terms of [J4].

2.1.3 Mapping (traceability) metamodels

In many model transformation approaches, source and target metamodels are complemented with another metamodel, which specifies the interconnections allowed between elements of the source and target languages. This metamodel has various names. In triple graph grammars [124], it is called a *correspondence metamodel*, while it is also called a *weaving metamodel* in [22, 46]. In previous works of the authors, the term *reference metamodel* was used [K23], which unfortunately coincides with reference models introduced in [78]. The same concept is also frequently called as *traceability metamodel*.

In order to avoid unintended clashes in terminology, in the current thesis, we refer to this metamodel as **mapping metamodel** (or **traceability metamodel** if we aim to emphasize its traceability role). Instances of this metamodel are called **mapping models** (or **traceability models**).

We require that all edges in the mapping metamodel (i.e. those associations that lead out from classes of the mapping metamodel) have an at most one multiplicity.As a consequence, each mapping node in a model uniquely identifies all the interconnected elements in the source and target models.

Note that a traceability (or mapping) model records the interconnections between models of different modeling languages while an execution trace model records the execution trace of a single (dynamic) model.

2.1.4 Formalizing models and metamodels

The metamodels of different modeling languages and their instance models are frequently formalized as type graphs and instance graphs are typed over this type graph [37]. The traditional instance-of relationship between metamodels and models is captured formally by a typing morphism.

Definition 2.1 (Graph) A **graph** $G = (N, E, src, trg)$ is a 4-tuple with a set N of nodes, a set E of edges, a source and a target function $src, trg : E \to N$.

Definition 2.2 (Type and instance graphs) A **type graph** TG is an ordinary graph. An **instance graph** G is typed over TG by a typing morphism $type : G \to TG$.

Definition 2.3 (Cardinality) Let $card(G, x)$ denote the **cardinality** (i.e. the number of graph objects) of a type $x \in TG$ in graph G. Formally, $card(G, x) = |\{n \mid n \in N \cup E \wedge type(n) = x\}|$.

For the current thesis, we assume that there is a unique edge of a certain type between two nodes, i.e., if $src(e_1) = src(e_2) \wedge trg(e_1) = trg(e_2) \wedge type(e_1) = type(t_2) \Rightarrow e_1 = e_2$, which simplifies the proofs of our theorems.

2.1.5 Example: Metamodels of the Object-Relational Mapping (ORM)

As the running example of the current thesis, we map (simplified) UML class diagrams into relational database tables by using one of the standard solutions [113, 131]. This transformation problem (with several variations) is frequently used as a model transformation benchmark of high practical relevance [5].

Source and target metamodels

The source and target languages (UML and relational databases, respectively) are captured by their respective metamodels in Fig. 2.2. To avoid mixing the notions of UML class diagrams and metamodels, we will refer to the concepts of the metamodel using nodes and edges (more precisely, node types and edge types) for classes and associations, respectively. Since model transformation rules will be formalized by graph transformation [119] (Sec. 2.2.2), this terminology is compliant with the notions of the model transformation domain.

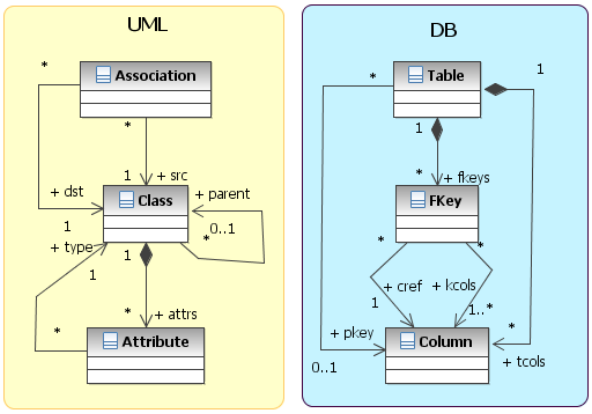


Figure 2.2: Source and target metamodels of the example

UML class diagrams consist of class nodes arranged into an inheritance hierarchy (represented by *parent* edges). Classes contain attribute nodes (*attrs*), which are typed over classes (*type*). Associations (graphically depicted by directed edges) are leading from a source (*src*) class to a destination (*dst*) class.

Relational databases consist of table nodes, which are composed of column nodes by *tcols* edges. Each table has a single primary key column (*pkey*). Foreign key (*FKey*) constraints can be assigned to tables (*fkeys*). Such a key refers to columns (*cref*) of another table, and it is related to the columns of local referring table by *kcols* edges.

Mapping metamodels of the ORM

In our running example (see Fig. 2.3), three types of mapping nodes are defined, namely, *Cls2Tab*, *Asc2Tab*, *Attr2Col*, which pair classes to tables, associations to tables and attributes to columns, respectively. In addition, various typed edges interconnect these mapping nodes to elements in the source and target metamodels.

- In case of *Cls2Tab*, edge *class* points to a *Class* in the source language, edge *tab* links to the corresponding *Table*, edge *pkey* marks the primary key *Column* in the database model, while *kind* denotes the *Column* storing the type information for the instances. As a notational shorthand, we may only depict the *tab* edge on the target side of *Cls2Tab* in various figures.

- In case of *Asc2Tab*, an edge *assoc* points to an *Association*, and a *tab* edge denotes the *Table* derived as main corresponding target element. In addition, a pair of edges (prefixed with *src* and *trg*, respectively) are used to mark the *Columns* storing the identifiers of classes and foreign key *FKey* restrictions (as denoted by edges *srccol*, *srckey*, *trgcol* and *trgkey*). As a notational shorthand, we may only depict the *tab* edge on the target side of *Asc2Tab* in various figures.

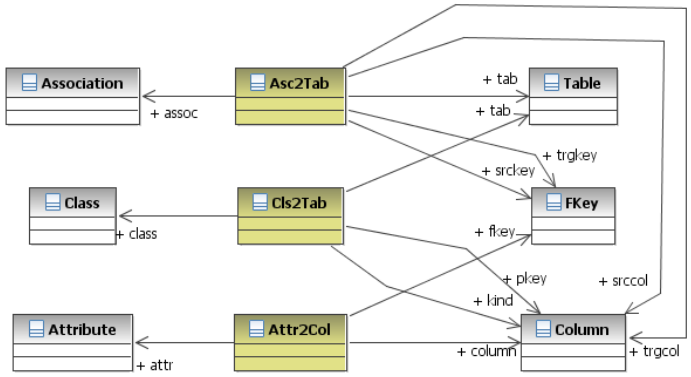


Figure 2.3: Mapping metamodel

- Finally, in case of *Attr2Col*, an edge *attr* links a source *Attribute* with a corresponding target *Column*. As a shorthand, we may only keep the *col* edge on the target side of *Attr2Col* in various figures.

For the sake of convenience, we assume that these (outgoing) edges of the mapping metamodel can be (totally) ordered for each type of mapping nodes and edges pointing to source elements precede edges leading to target elements. For instance, we can assume that the edges of the *Cls2Tab* node are ordered in the following way: *class*, *tab*, *pkey* and *kind*.

Definition 2.4 (Traceability relations) Traceability relations enable to represent (model-level) mapping structures as tuples

$$ref(r_i, src_1, \ldots, src_n, trg_1, \ldots, trg_m)$$

where ref corresponds to the type of the mapping node, r_i is the unique identifier of the node, $src_1, \ldots, src_n$ are nodes of the source model (defined by the order of mapping edges), while $trg_1, \ldots, trg_m$ are nodes of the target model linked by appropriate mapping edges.

2.2 Model Transformations by Graph Transformation

We first overview the main concepts of model transformations, and then summarize the formal paradigm of graph transformation, which is frequently used as a means to precisely capture model transformations.

2.2.1 Overview of Model Transformations

Classification of model transformations

Model transformations can be categorized [40] as **in-place transformations** versus **inter-model transformations**, where the former involves transformations over a single modeling language, while the latter carries out transformations from one (or more) source language to one (or more) target language.

A model transformation can be **unidirectional** or **bidirectional**. A unidirectional transformation can be executed only in one way (i.e. to generate the target equivalent of a source

model), while a bidirectional transformation is executable in both forward and backward directions (from source to target and also from target to source, respectively). Most model transformations between dynamic modeling languages carry out powerful abstractions, therefore, they are unidirectional by nature. This way, the primary focus of the thesis is on unidirectional transformations.

A **source incremental model transformation** aims to avoid unnecessary model traversal, and only reads those parts of the model which are relevant for change detection and propagation. The design goal of source incrementality is to minimize the query overhead, since the execution of model queries is computationally expensive. A **target incremental model transformation** does not re-generate the target model from scratch, but updates the models instead.

Operational semantics and traces for dynamic models

The simulation / execution of a DML is performed in accordance with the **operational semantics** of the language defined by simulation rules. In our framework we assume that simulation rules are defined as in-place model transformations illustrated in Fig. 2.4 (see also [43,52,K19]).

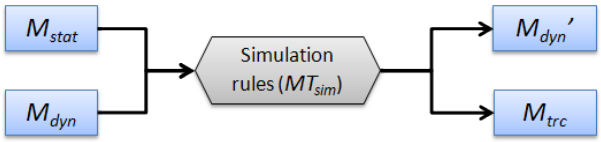


Figure 2.4: Model transformations for operational semantics and execution trace generation

The **execution of a transformation rule** $MT_{sym} : (M_{stat}, M_{dyn}) \rightarrow \Delta M_{dyn}'$ modifies the M_{dyn} by also taking into account M_{stat} and results in a new M_{dyn}'. During a simulation run, the changes of the dynamic model are recorded as a sequence of micro steps as part of the derived trace model M_{trc}.

An alternate approach is to provide semantics to a DML in a denotational way, which is investigated in depth by the semantic anchoring approach proposed in [31, 32].

Forward model transformation

A unidirectional forward model transformation $MT_{src2trg}$ (see Fig. 2.5) generates a static target model M^{trg}_{stat} from a given static source model M^{src}_{stat}. This MT is also responsible for deriving the initial state of M^{trg}_{dyn} from the initial state of M^{src}_{dyn} in case of a DML.

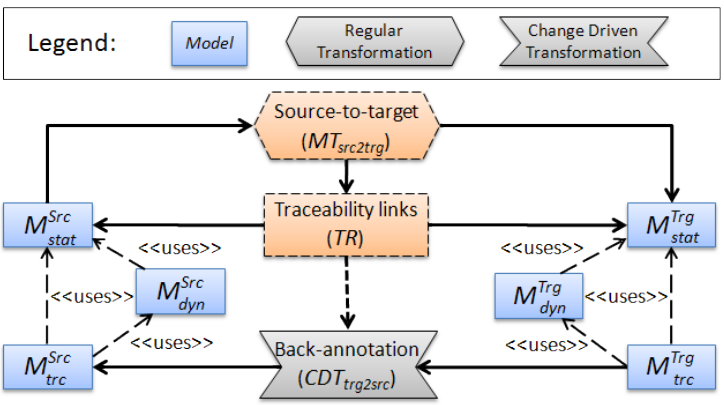


Figure 2.5: Forward model transformation and back-annotation

This transformation also generates the mapping (traceability) models (TR) between the source and target models in order to record the structural correspondence between the model elements. These traceability models can be used efficiently in various scenarios including incremental model synchronization, back-annotation, etc. as well as for certification purposes.

Forward unidirectional model transformations frequently automate the mapping from high-level design models into different analysis models in order to carry out formal verification and validation (V&V) by back-end analysis tools. Since formal analysis models are derived by automated model transformations, and the target analysis tools are also typically fully automated, systems and services engineers obtain a push-button technique for formally analyzing their high-level design models.

Back-annotation

Back-annotation aims at automatically mapping back the results of V&V tools to the original design model in order to highlight the real source of the flaw. In case of transformation between static modeling languages, back-annotation should simply identify the origins of a target model element in the source model.

The back-annotation of a target DML to a source DML is defined as a transformation $CDT_{trg2src}$ which is able to generate the M^{src}_{trc} from an arbitrary M^{trg}_{trc} if such source trace exists. The $CDT_{trg2src}$ makes use of the TR to identify corresponding elements in source and target models. As the traces contain model changes, in [K13], we proposed to define the $CDT_{trg2src}$ as a change driven model transformation [J4, K20].

2.2.2 Graph transformation

Definition of a graph transformation rule

Graph transformation (GT) [38, 51, 119] offers a rule and pattern based formal paradigm for precisely capturing the transformation of graph-based models.

Definition 2.5 (Graph transformation rule) A **graph transformation rule** $r = (L \xleftarrow{l} K \xrightarrow{r} R)$ typed over a type graph TG is given by triple where L (left-hand side, LHS), K (context) and R (right-hand side, RHS) graphs are typed over TG and graph morphisms l, r are injective and assumed to be type preserving. Graphs L and R are frequently called as **graph patterns**.

Definition 2.6 The **negative application conditions** (NACs) of a GT rule are a (potentially empty) set of pairs (N, n) with N being a graph also typed over TG and $n : L \to N$ being an injective graph morphism. A GT rule with NACs is denoted shortly as $r = (L \xleftarrow{l} K \xrightarrow{r} R, \{L \xrightarrow{n_i} N^i\})$ $(i = 1 \dots k)$. Moreover, we assume that no rules exist where all L and N are empty.

The L and the N graphs are together called the precondition PRE of the rule.

Application of a graph transformation rule.

Next we describe how a graph transformation rule can be applied (executed, fired) in order to transform a graph G into a result graph H.

Definition 2.7 (Application of a GT rule) The **application of a GT rule** to a *host model graph* G alters the model graph by replacing the pattern defined by L with the pattern defined by R.

This is performed by

1. *finding an injective match* $m : L \to G$ of the L pattern in the model graph G;

2. *checking the negative application conditions* N which prohibit the presence of certain model elements, i.e. for each NAC $n : L \to N$ of a rule no injective graph morphism $q : N \to G$ exists with $m = q \circ n$;

3. *removing* a part of the model graph M that can be mapped to L but not to R yielding an intermediate graph D;

4. *adding* new elements to the intermediate graph D which exist in R but not in L yielding the derived graph H.

Steps 1 and 2 above are called **graph pattern matching**.

A **GT step** is denoted formally as $G \stackrel{r,m}{\Longrightarrow} H$, where r and m denote the applied rule and the match along which the rule was applied, respectively.

Definition 2.8 (Double Pushout Approach) The **Double Pushout Approach** [38] conservatively refines the above semantics of graph transformation rules by introducing the *dangling condition* to claim that a node cannot be deleted if it has connected edges which are not explicitly deleted by the rule. Furthermore, the *identification condition* states that only rule objects in L prescribing deletion (i.e. in L but not in K) need to be matched injectively.

Definition 2.9 (Single Pushout Approach) The **Single Pushout Approach** [53] greedily removes all the dangling edges, i.e. the removal of a node implicitly removes all the related edges.

Example 2.10 A sample graph transformation rule calculating the transitive closure of the *parent* relation is depicted in the top rule (*parentClosureR*) of Fig. 2.6. The rule prescribes that if class *CP* is parent of class *CM* (i.e. there is a *parent* edges between them), and *CM* is a parent of class *CC*, but there is no *parent* edge from *CC* to *CP*, then such an edge should be created.

For a more compact presentation of the rules, we abbreviate the L, N and R graphs of a rule into one, and we only mark the (images of) graph elements to be removed (*del*), or created (*new*). We assume that all elements in R marked as *new* are implicitly present in the negative application condition N as well. In case of rule *class2TableR* we use crossed lines to denote the second negative application condition (that is not part of R).

State space of a graph grammar

Definition 2.11 (Graph transformation system) A **graph transformation system** $GTS = (R, TG)$ consists of a type graph TG and a finite set R of graph transformation rules typed over TG.

Definition 2.12 (Graph grammar) A **graph grammar** $GG = (GTS, G_0)$ consists of a graph transformation system $GTS = (R, TG)$ and a so-called *start (model) graph* G_0 typed over TG.

Definition 2.13 (State space of a graph grammar) The **state space** $Sem(GG)$ generated by a graph grammar $GG = (GTS, G_0)$ is defined as a graph where nodes are model graphs, and edges are graph transformation steps $G \stackrel{r,m}{\Longrightarrow} H$ such that the source and target nodes of the edge are graphs G and H, respectively. Starting from G_0 the state space (i.e. the reachable model graphs) of the GG is represented taking into account all applicable rules from a given model graph for all possible matches.

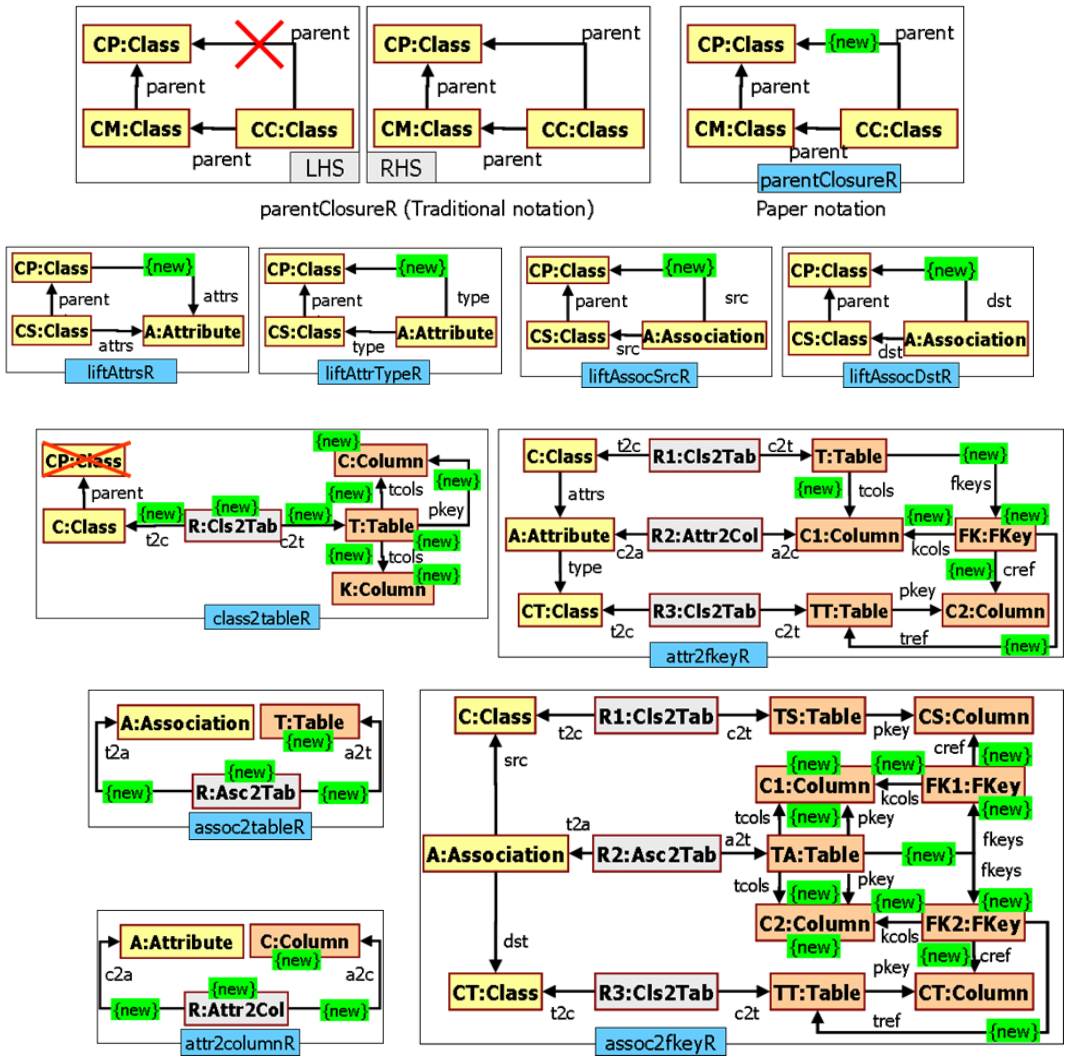


Figure 2.6: Model transformation from UML to relational databases

Definition 2.14 (Termination) A *graph grammar* $GG = (G_0, GTS)$ is *terminating* if there are no infinite sequences of rule applications starting from G_0. A *graph transformation system* $GTS = (R, TG)$ is called *terminating* if for all G_0, the corresponding graph grammar $GG = (G_0, GTS)$ is terminating.

2.2.3 Example: The Object-Relational Mapping as a model transformation problem

The object-relational mapping as a model transformation problem can be summarized as follows:

- Each top-level UML class (i.e. a top-most class in the inheritance tree) is projected into a database table. Two additional columns are derived automatically for each top-level class: one for storing a unique identifier (primary key), and one for storing the type information of instances.
- Each attribute of a UML class will appear as columns in the table related to the top-level ancestor of the class. For the sake of simplicity, the type of an attribute is restricted to user-defined classes. The structural consistency of valid object instances in columns is maintained by foreign key constraints.

- Each UML association is projected into a table with two columns pointing to the tables related to the source and the target classes of the association by foreign key constraints.

The object-relational mapping is captured by the set of graph transformation rules in Fig. 2.6. The entire transformation starts with a preprocessing phase when the transitive closure of *parent* relations is calculated (*parentClosureR*), and then all attributes and associations are lifted up to the top-level classes in the inheritance (*parent*) hierarchy (rules *liftXYZ*). Then the main model transformation (Fig. 2.6) proceeds by transforming classes into tables (*class2tableR*), associations into tables (*assoc2tableR*), attributes into columns (*attr2columnR*), attribute types and destination class of associations into foreign key constraints (*attr2fkeyR* and *assoc2fkeyR*).

From a pure functional point of view, this solution appropriately addresses the object-relational mapping as a model transformation problem. However, the transformation has side effects on the source UML model by lifting *parent*, *type*, *src* and *trg* edges. In many model transformation scenarios, such side effects are disallowed. The core graph transformation formalism does not support more advanced graph queries, which could be used to prevent this issue. The graph pattern based query language to be introduced in Chapter 3 will address this problem.

2.3 An Overview of Inductive Logic Programming

Inductive logic programming [101] (ILP) aims at inductively constructing first-order clausal hypotheses from examples and background knowledge. In case of ILP, induction is typically interpreted as abduction combined with justification. **Abduction** is the process of hypothesis formation from some facts while *justification* (or confirmation) denotes the degree of belief in a certain hypothesis given a certain amount of evidence. Formally, the problem of **inductive inference** can be defined as follows [101].

Definition 2.15 (Inductive inference) Given some (a priori) background knowledge B together with a set of positive facts E^+ and negative facts E^-, find a hypothesis H such that the following conditions hold:

Prior Satisfiability. All $e \in E^-$ are false in $\mathcal{M}^+(B)$ where $\mathcal{M}^+(B)$ denotes the minimal Herbrand model of B (denoted as $B \wedge E^- \not\models \top$).

Posterior Satisfiability (Consistency). All $e \in E^-$ are false in $\mathcal{M}^+(B \wedge H)$ (denoted as $B \wedge H \wedge E^- \not\models \top$).

Prior Necessity. E^+ is not a consequence of B, i.e. some $e \in E^+$ are false in $\mathcal{M}^+(B)$ (denoted as $B \not\models E^+$).

Posterior Sufficiency (Completeness). E^+ is a consequence of B and H, i.e. all $e \in E^-$ are true in $\mathcal{M}^+(B \wedge H)$ (denoted as $B \wedge H \models E^+$).

A generic ILP algorithm [101] keeps track of a queue of candidate hypotheses. It repeatedly selects (and removes) a hypothesis from the queue, and expands that hypothesis using inference rules. The expanded hypothesis is then added to the candidate queue, which may be pruned to discard unpromising hypotheses from further consideration.

Many existing ILP implementations like Aleph [3] that we used for our experiments are closely related to Prolog, and the following restrictions are quite typical:

- B is restricted to definite clauses where the conjunction of (positive or negative) body clauses implies the head, formally

$$Head : -Body_1, Body_2, \ldots Body_n$$

- E^+ and E^- are restricted to ground facts.

Further language and search restrictions can be defined in Aleph by using mode, type and determination declarations. Moreover, we can also ask in Aleph to find all negative constraints, i.e. Prolog clauses of the form *false :-* $Body_1$, $Body_2$,...,$Body_n$, More details on negative constraints will be given in Sec. 4.4.3.

Example 2.16 As a demonstrative example, let us consider some traditional family relationship. The background knowledge B may contain the following clauses:

```
grandparent(X,Y) :- father(X,Z), parent(Z,Y).
father(george,mary).
mother(mary,daniel).
mother(mary,greg).
```

Some positive examples E^+ can be given as follows:

```
grandfather(george,daniel).
grandfather(george,greg).
```

Finally, some negative facts E^- are also listed:

```
grandfather(daniel,george).
grandfather(mary,daniel).
```

Believing B, and faced with the facts E^+ and E^-, Aleph is able to set up the following hypothesis H by default.

```
grandfather(X,Y) :- father(X,Z), mother(Z,Y).
```

By default settings, Aleph will set up a hypothesis only for clauses used in the positive and negative examples. However, there is some (limited) support for abductive learning [6] when Aleph will be able to derive hypothesis not only for the clause covered by positive and negative examples:

```
parent(X,Y) :- mother(X,Y).
```

The model transformation by example approach to be presented in Chapter 4 relies upon Aleph, which is considered to be a powerful inductive logic programming engine.

2.4 An Introduction to Petri Nets

In the current section we give a short introduction into the theory of Place/Transition nets based on [102].

Definition 2.17 (Place/Transition (P/T) Net) A **Place/Transition net** (or shortly P/T net or Petri net) is a 4-tuple $PN = (P, T, E, w)$ where P is a set of **places** (represented graphically as circles), T is a set of **transitions** (represented as horizontal bars), $E \subseteq (P \times T) \cup (T \times P)$ is the set of **arcs** (where no arcs connect two places or two transitions), and the **weight function** $w : E \to \mathbb{N}^+$ maps arcs to positive integers.

Places may contain tokens. The distribution of tokens at different places is called a **marking** $M : P \to \mathbb{N}$, which maps places to non-negative integers. The initial marking is denoted as M_0.

Definition 2.18 (Firing a transition) The token distribution can be changed in the net by firing transitions. A transition t is **enabled** (i.e. it may fire), if each of its input places contain at least as many tokens as it is specified by the weight function. The **firing** of an enabled transition t removes a $w(p,t)$ amount of tokens from the input places, and $w(t,p)$ tokens are produced on each output place p. As a result, the marking M changes to M' (denoted as $M \stackrel{t}{\Longrightarrow} M'$) according to $\forall p \in P : M'(p) = M(p) - w(p,t) + w(t,p)$.

Definition 2.19 (Incidence matrix) The **incidence matrix** W of a (finite) net describes the net token flow (of the P/T net) when firing a transition. Mathematically, W is a $|P| \times |T|$–dimensional matrix of non-negative integers $\mathbb{N}$ such that $w_{ij} = w(t_j, p_i) - w(p_i, t_j)$, where $1 \leq i \leq |P|, 1 \leq j \leq |T|$.

After firing a transition t in marking M, the result marking M' can be computed with the incidence matrix: $M' = M + W \cdot \underline{e}_t$, where $\underline{e}_t$ is a $|T|$-dimensional unit vector, where the t-th component is 1 and the others are 0.

Definition 2.20 (firing sequence) A **(transition) firing sequence** $s = \langle t_1, t_2, \ldots, \rangle$ is a sequence of transition firings starting from state M_0 such that $M_0 \stackrel{t_1}{\Longrightarrow} M_1, \stackrel{t_2}{\Longrightarrow} \ldots$, i.e. for all $1 \leq j$ t_j is enabled in M_{j-1} and M_j is yielded by the firing of t_j in M_{j-1}.

Definition 2.21 (Transition occurrence vector) The marking of the net after executing the first k steps of the firing sequence s can be calculated by the state equation: $M_k = M_0 + W \cdot \underline{\sigma}$, where $\underline{\sigma}$ is the **transition occurrence vector** or **Parikh–vector** of the trajectory s counting the number of occurrences of individual transitions in the firing sequence.

2.5 SUMMARY

In this chapter, we introduced the core software engineering concepts of domain-specific modeling languages, metamodels and model transformations by using the formal paradigm of typed graphs and graph transformations. These concepts will be used intensively in all upcoming chapters of the current thesis. Furthermore, inductive logic programming will be exploited in Chapter 4, while Petri nets will be used in Chapter 6.

Chapter 3

Specification of Model Transformations

3.1 Introduction

The crucial role of model transformation (MT) languages and tools for the overall success of model-driven system development has been revealed in many surveys and papers during the recent years. To provide a standardized support for capturing queries, views and transformations between modeling languages defined by their standard MOF metamodels, the Object Management Group (OMG) has recently issued the QVT standard [108].

QVT provides an intuitive, pattern-based, bidirectional model transformation language, which is especially useful for synchronization kind of transformations between semantically equivalent modeling languages. Unfortunately, QVT is not a mathematically precise language, several semantic gaps and choice points have been identified in [67].

Model transformations used for model validation [24] simultaneously necessitate a specification language with intuitive syntax and formal semantics. Therefore, we have chosen to integrate two intuitive, and mathematically precise rule-based specification formalisms, namely, graph transformation (GT) [51] and abstract state machines (ASM) [25] to manipulate graph based models. This forms the specification language of the VIATRA2 framework.

VIATRA2 also facilitates the separation of transformation design (and validation) and execution time. During design time, the VIATRA2 interpreter takes such MT descriptions and executes them on selected models as experimentation. Then final model transformation rules can be compiled into efficient, platform-specific transformer plugins for optimal execution [K2] (see later in Chapter 5).

In the current chapter, we present the VIATRA2 transformation language which is composed of three sublanguages for metamodeling (Sec. 3.2), pattern specification (Sec. 3.3), and rule-based model transformations (Sec. 3.4). The current chapter includes a detailed description of the ASM control language (Sec. 3.4.3) and the formal semantics of the major VIATRA2 constructs (Sec. 3.4.2 and Sec. 3.6). Reusability is also addressed by introducing generic rules in Sec. 3.5.1. Finally, Sec. 4.7 discusses related work to justify that VIATRA2 was first to introduce recursive graph patterns with arbitrary negation depth, and generic transformations.

3.2 Metamodeling Language

Metamodeling is a fundamental part of model transformation design as it allows the structural definition (i.e. abstract syntax) of modeling languages. Metamodels are represented in a meta-

modeling language, which is another modeling language for capturing metamodels.

In the current chapter, we build upon the VPM metamodeling framework [J15] for model representation and management. As only an abstract syntax was defined for VPM in [J15], we developed a textual concrete syntax for the metamodeling environment called *VTML (Viatra Textual Metamodeling Language)* for specifying metamodels and models. The syntax of the VTML language has a certain Prolog flavor, but it offers support for well-founded typing and hierarchical model libraries. Our experience showed that this textual format was more usable in case of large, or automatically generated metamodels compared to a graphical language.

Standard metamodeling paradigms and languages can be integrated into VIATRA2 by import plugins and exporters defined by code generation transformations. So far, models from very different technological spaces such as XML, EMF, semantic web and modeling languages of high practical relevance like BPEL, UML (and various domain-specific languages in the dependable embedded, telecommunication, and service-oriented domain) have been successfully integrated into VIATRA2 . While VIATRA2 offers the VTML language for constructing models and metamodels, the main usage scenario is to bridge heterogeneous off-the-shelf tools by flexible model imports and exports.

3.2.1 Metamodeling concepts in VTML

Entities and relations. As shown in the metamodel of Fig. 3.1, the VPM model space consists of two basic elements: the entity (a generalization of MOF package, class, or object) and the relation (a generalization of MOF association end, attribute, link end, slot). **Entities** represent basic concepts of a (modeling) domain, while **relations** represent the relationships between model elements. Furthermore, entities may also have an associated string value which contains application-specific data. Most typically, relations lead between two entities to impose a directed graph structure on VPM models, but the source and/or the target end of relations can also be relations, thus hypergraphs can also be represented as VPM models.

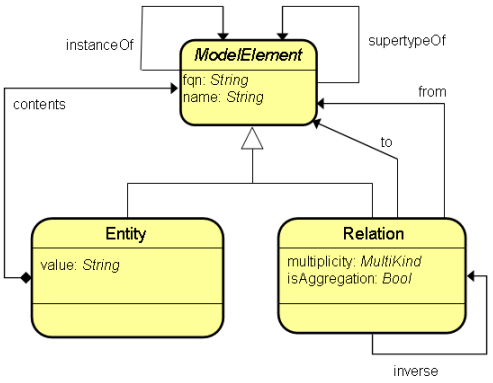


Figure 3.1: The metamodel of the VPM model space

Relations have additional properties. (i) Property **isAggregation** tells whether the given relation represents an aggregation in the metamodel, when an instance of the relation implies that the target element of the relation instance also contains the source element. (ii) The **inverse** relation points to the inverse of the relation (if any). In a UML analogy, a relation can be considered as an association end, and thus the inverse of a relation denotes the other association end along an association. (iii) **Multiplicities** impose restrictions on the model structure. Allowed multiplicities in VPM are *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many*.

In VTML, an entity can be declared in the form type(name), where *type* is the type of the entity and *name* is a fresh (unused) name for the new entity. A relation definition takes the

form: type(name, source, target), where *source* and *target* are either existing elements in the model space, or they are defined (or to be defined) in the same VTML file. Type declarations are mandatory for all model elements with the basic VPM *entity* and *relation* model elements in the top of the type hierarchy.

Containment hierarchy and namespace imports. Model elements are arranged into a strict **containment hierarchy**. Within a container entity, each contained model element has a unique local name. In addition, a globally unique identifier called a **fully qualified name (FQN)** is defined for each model element. An FQN is derived by concatenation along the containment hierarchy of local names using dots (".") as separators. Relations are automatically assigned to their source model element within this containment hierarchy, which corresponds to the design decision in many modeling tools. For example, the FQN of the entity Association in Fig. 3.3 is *UML.Association*, while the FQN of the relation *src* is *UML.Association.src*.

VTML also allows to **import namespaces** from the model space for the given VTML file so that model elements in the imported namespaces can be referred to using their local names instead of their FQNs.

Inheritance and instantiation. There are two special relationships between model elements (i.e. either between two entities or two relations): the **supertypeOf** (inheritance, generalization) relation represents binary superclass-subclass relationships (like the UML generalization concept), while the **instanceOf** relation represents type-instance relationships (between metalevels). By using explicit *instanceOf* relationship, metamodels and models can be stored in the same model space in a compact way. Note that both multiple inheritance and multiple typing are allowed.

Finally, it is worth pointing out that many advanced modeling concepts of MOF 2.0 can be easily modeled in VPM. For instance, ordered properties can be represented as (ordering) relations leading between relations, while the subset property can be modeled by *supertypeOf* relationship between the two relations representing the properties.

3.2.2 Demonstrating example

The technicalities of VTML are demonstrated in Fig. 3.2 using the metamodels of a model transformation from UML class diagrams to description logic. This transformation was implemented in the context of the DECOS European IP [73, 122] to carry out semantic validation of domain-specific modeling languages and models in the dependable embedded domain using the Racer reasoner [114]. Extracts from this transformation (with a simplified UML metamodel and a subset of rules) will be used in this chapter for demonstration purposes.

The UML dialect used in this chapter contains *Classes*, which may have attributes typed by a *Classifier* which is either a *PrimitiveDataType* or by other user defined *Classes*. Furthermore, *Associations* may lead from a source (*src*) class to a target (*dst*) class.

The metamodel of the Racer tool consists of *Concepts*, which can be interrelated through several *Roles* (as indicated by *domain* and *range*). Each concept is allowed to have several attributes (*Attribute*) denoted by *attr*. Inheritance between concepts (similar to UML) can be defined using *Implication* elements. A child concept (denoted by *impl*) implicates its super concept (*subject*).

Finally, the source and target metamodels are interrelated by using the constructs of a mapping metamodel (denoted as *Ref*). This mapping metamodel declares that a UML class may be related to a Racer concept (as *cls2concept*), a UML association may also be connected to

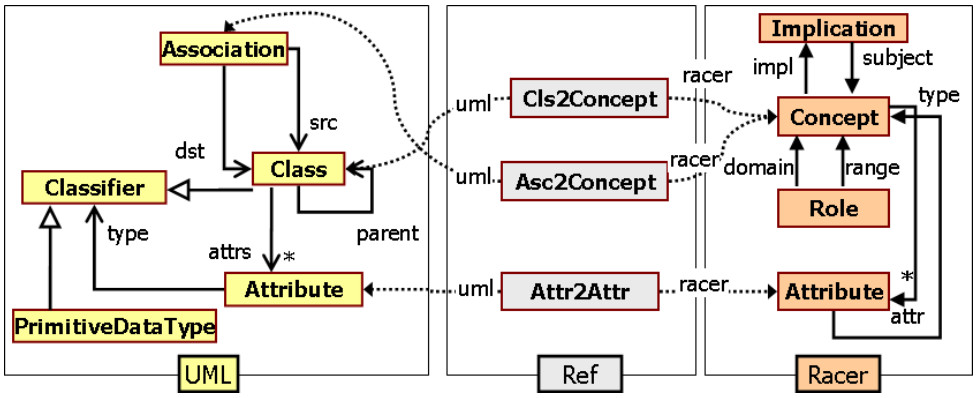


Figure 3.2: Sample UML and Racer metamodels

a Racer concept (as *asc2concept*), and UML attributes can be related to Racer attributes (see *att2attr*). The VTML representation of the metamodels is listed in Fig. 3.3.

```
entity(UML)
{
 entity(Classifier);
 entity(PrimitiveDataType);
 supertypeOf(Classifier,
              PrimitiveDataType);

 entity(Class);
 supertypeOf(Classifier,Class);
 relation(parent,Class,Class);

 entity(Association);
 relation(src,Association,Class);
 relation(dst,Association,Class);

 entity(Attribute);
 relation(type,Attribute,
               Classifier);
 relation(attrs,Class,Attribute);

 multiplicity(attrs,many-to-many);
 isAggregation(attrs, true);

}
```

```
entity(racer)
{
 entity(concept);
  relation(attr,concept,attribute);
  relation(impl,concept,implication);
 entity(role);
  relation(domain,role,concept);
  relation(range,role,concept);
 entity(attribute);
  relation(type,attribute,concept);
 entity(implication);
  relation(subject,implication,concept);
}
entity(Ref) {
 entity(class2concept);
  relation(uml,class2concept,Class);
  relation(racer,class2concept,concept);
 entity(assoc2concept);
  relation(uml,assoc2concept,Association);
  relation(racer,assoc2concept,concept);
 entity(attr2attr);
  relation(uml,attr2attr,Attribute);
  relation(racer,attr2attr,attribute);
}
```

Figure 3.3: Sample VTML metamodels: UML and Racer

3.3 The Pattern Language

Graph patterns (GP) are the atomic units in VIATRA2 for capturing well-formedness rules of a modeling language and, especially, for specifying common patterns used in model transformation rules. Graph patterns are integral part of the *Viatra Textual Command Language (VTCL)*, which is the main textual language in VIATRA2 to specify model transformations.

3.3.1 Graph patterns

Graph patterns represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. A model (i.e. part of

the model space) satisfies a graph pattern, if the pattern can be matched to a subgraph of the model using *graph pattern matching*. In this section, we present an informal introduction to graph patterns in VTCL, while their formal semantics will be discussed in Sec. 3.6.

Simple patterns, negative patterns

Patterns are close to predicates in Prolog, as they have a name, a parameter list, and a body. The body of a simple pattern contains model element and relationship definitions using VTML language constructs.

In the left column of Fig. 3.4, a simple pattern can be fulfilled by a class which has an attribute with a user-defined type. Here *Class(C)* declares a variable *C* to store an entity of type *Class* while *Class.attrs(X,C,A)* denotes a relation of type *Class.attrs*, which has an identifier *X*, and leads from class *C* to attribute *A*. Note that these predicates can be listed in an arbitrary order (unlike in Prolog), i.e. the transformation engine is responsible for the appropriate ordering of predicates by using sophisticated algorithms for search plan generation [J20].

```
import UML;

/*C is a UML class with an
  attribute A; A is of type T*/

pattern isClassAttribute(C, A) =
{
 Class(C);
//X is the relation between C and A
//it is an internal variable, it is
//not present in the interface
 Class.attrs(X,C,A);
 Attribute(A);
 Attribute.type(Y,A,T);
 Classifier(T);
}
```

```
import UML;
/* C is a class without parents
     and with non-empty name */
pattern isTopClass(C, M) =
{
 Class(C) below M;

 neg pattern negCondition(C) =
 {
   Class(C);
   Class.parent(P,C,CP);
   Class(CP);
 }

 check (name(C)!="")
}
```

Figure 3.4: Basic patterns and negative patterns

The keyword *neg* denotes if a subpattern serves as a negative condition for another pattern. The negative pattern in the right column of Fig. 3.4 can be satisfied if there is a class (*CP*) for the class in the parameter (*C*) that is the parent of C as indicated by relation *P* of type *Class.parent*. If this condition can be satisfied, the outer (positive) pattern matching will fail. Thus the pattern matches to top-most classes in the *parent* hierarchy.

Each entity in a pattern may be *scoped* by using the *in* or *below* keywords by a container entity. This means that the corresponding pattern element should be matched to a model element which resides directly inside (*in*) or somewhere below (*below*) its scope entity in the containment hierarchy of the model space. Additional Boolean constraints can be expressed by the *check* condition, which also need to be fulfilled for successful pattern matching. Our sample pattern *isTopClass* can be matched to classes with non-empty names.

A VTCL pattern language allows that negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations) when the expressiveness of such patterns converges to first order logic [115].

Pattern calls, OR-patterns, recursive patterns

In VTCL, a pattern may call another pattern using the *find* keyword. This feature enables the reuse of existing patterns as a part of a new (more complex) one. The semantics of this reference

is similar to that of Prolog clauses: the caller pattern can be fulfilled only if their local constructs can be matched, and if the called (or referenced) pattern is also fulfilled.

Alternate bodies can be defined for a pattern by simply creating multiple blocks after the pattern name and parameter definition, and connecting them with the *or* keyword. The OR-pattern is fulfilled if at least one of its bodies can be fulfilled. As OR-patterns can be called from other patterns, thus, allowing disjunction only on the top-level is not a real limitation.

Pattern calls and alternate (OR) bodies can be used together for the definition of *recursive patterns*. In a typical recursive pattern, the first body (or bodies) define the halt condition for the recursion, while subsequent bodies contain a recursive call to itself. However, VIATRA2 supports general recursion, i.e. multiple recursive calls are allowed from a pattern. Note that general recursion is not supported by any of the existing graph transformation tools up to now. The following example in Fig. 3.5 illustrates the usage of recursion.

```
// Parent is an ancestor (transitive parent) of Child
pattern ancestorOf(Parent,Child) =
{
   Class(Parent);
   Class.parent(X,Child,Parent);
   Class(Child);
}
or
{
   Class(Parent);
   Class.parent(X,C,Parent);
   Class(C);
   find ancestorOf(C,Child);
   Class(Child);
}
```

Figure 3.5: Recursive patterns

A class *Parent* is the ancestor of an other class *Child*, if *Child* is a direct child of classes *Parent*, or *Parent* has a direct child (C), which is the parent of the child class (second body). The pattern uses recursion for traversing multi-level parent-child relationships, and uses multiple bodies to create a halt condition (base case) for the recursion.

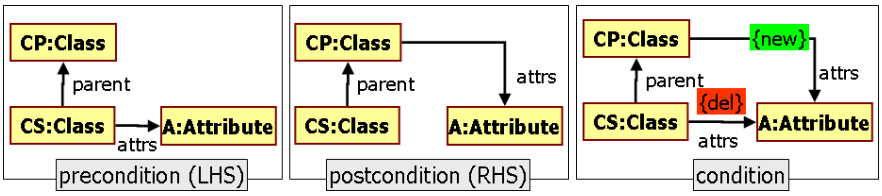
3.4 THE TRANSFORMATION LANGUAGE

Transformation descriptions in VIATRA2 consist of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph transformation (GT) [51] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [25] rules can be used for the description of control structures.

3.4.1 Graph transformation rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules [51], which heavily rely on graph patterns for defining the application criteria of transformation steps. The application of a GT rule on a given model replaces an image of its left-hand side (LHS) pattern with an image of its right-hand side (RHS) pattern. The VTCL language allows different notations for defining graph transformation rules using the *gtrule* keyword.

The sample graph transformation rule in Fig. 3.6 defines a refactoring step of lifting an attribute from child to parent classes (repeated from Fig. 2.6). This means that if the child class has an attribute, it will be lifted to the parent.



(a) Traditional notation (b) Compact notation

Figure 3.6: Sample graph transformation rule

The first syntax of a GT rule corresponds to the traditional notation (see the graphical notation in Fig. 3.6(a) and the textual one in Fig. 3.7). It contains a **precondition** pattern for the LHS, and a **postcondition** pattern that defines the RHS of the rule. In general, elements that are present only in (the image of) the LHS are deleted, elements that are present only in RHS are created, and other model elements remain unchanged. Moreover, further actions can be initiated by calling any ASM rules within the **action** part of a GT rule, e.g. to report debug information or to generate code. This *action* part is executed *after* the model manipulation part is carried out according to the difference of the precondition and postcondition part.

```
import UML;
gtrule liftAttrsR(inout CP, inout CS, inout A) =
{
   precondition pattern lhs(CP,CS,A,Par,Attr) =
   {
      Class(CP);
      Class.parent(Par,CS,CP);
      Class(CS);
      Class.attrs(Attr,CS,A);
      Attribute(A);
   }
   postcondition pattern rhs(CP,CS,A,Par,Attr,Attr2) =
   {
      Class(CP);
      Class.parent(Par,CS,CP);
      Class(CS);
      Class.attrs(Attr2,CP,A);
      Attribute(A);
   }
   action {
      print("Rule liftAttrR is applied on attribute " + name(A));
   }
}
```

Figure 3.7: Graph transformation rule in traditional notation

Negative conditions are also commonly used in precondition patterns, especially, for model transformations between modeling languages in order to prevent the application of a GT rule twice on the same match. Examples for negative application conditions will be given in the case study of Sec. A.1.

3.4.2 Informal semantics of graph transformation rules

Parameter passing. A main difference with the traditional GT notation is related to the use of parameter passing between preconditions and postconditions. More precisely, matches of the precondition pattern are passed to the postcondition via pattern parameters, which act as an explicit interface between the precondition and the postcondition.

- A parameter of the postcondition is treated as an **input parameter** if (i) it is also a precondition parameter or (ii) it is passed to the entire GT rule as an input parameter. Note that a simple lexical match decides if a precondition parameter also appears as a postcondition parameter regardless of the order of parameters. These parameters are already bound before calculating the effects of the postcondition. In our example, input parameters of the postcondition are *CP*, *CS*, *A*, *Par* and *Attr*.

- Additional parameters of the postcondition are **output parameters**, which will be bound as the direct effect of the postcondition. The single output parameter of the postcondition of our example is *Attr2*.

On the one hand, we can reduce the size of the patterns with respect to traditional GT rules by information hiding. For instance, precondition elements which are left unchanged by the rule does not need to be passed to the postcondition, which is very convenient for large patterns.

The negative side of this solution is that the execution mechanism of a GT rule becomes slightly more complex than in case of traditional GT rules. More specifically, the postcondition of a GT rule may prescribe three different operations on the model space.

- *Preservation.* If an input parameter of the postcondition appears in the pattern body, then the matching model element is preserved. The elements matched by variables *CP*, *CS*, and *A* are thus preserved above.

- *Deletion.* If an input parameter of the postcondition does not appear in the body of the postcondition pattern then the corresponding model element is deleted. For instance, element matched by variable *Attr* is deleted.

- *Creation.* If a variable which appears in the body of the postcondition pattern is not an input parameter of the postcondition, then a new model element is created, and the variable is bound to this new model element. In our example above, variable *Attr2* is not an input parameter of the postcondition, thus it prescribes the creation of a new *attrs* relation between class *CP* and attribute *A*. This *Attr2* is an output parameter of the postcondition but not passed back to the GT rule itself.

This way, rule *liftAttrsR* can be further compacted by simply omitting the *parent* relation from the postcondition and the pattern parameter lists.

Pattern calls in GT rules. In order to reduce the size and to support the modular creation of GT rules, pattern calls (aka. pattern composition using the *find* construct) are allowed in both the precondition and postcondition pattern. Its use in the precondition pattern was already discussed in Sec. 3.3.1.

However, pattern calls in the postcondition (RHS) of GT rules is a unique feature compared to other GT tools. Currently, VIATRA2 handles non-recursive and non-negative calls in the postcondition pattern, which allows a macro-like substitution of the called pattern in the body

of the postcondition. This way, repetitive parts of a postcondition can be modularized into predefined patterns, which can be used in various GT rules afterwards. More examples of pattern calls will be provided in the case study of Sec. A.1.

Invoking graph transformation rules

The basic invocation of a graph transformation rule is initiated using the *apply* keyword within a *choose* or a *forall* construct (further details on these constructs are given in Sec. 3.4.3). In each case, the actual parameter list of the transformation has to contain a valid value for all input parameters, and an unbound variable for all output parameters.

A rule can be executed for all possible matches (in a single, pseudo-parallel step) by quantifying some of the parameters using the *forall* construct. Finally, a GT rule can be applied as long as possible by combining the *iterate* and the *choose* constructs. The example in Fig. 3.8 illustrates some possible invocations of our sample rule *liftAttrsR*.

```
//execution of a GT rule for one attribute of a class
//variables Class1 and Class2 must be bound
choose A apply liftAttrsR(Class1,Class2,A);

//calling the rule for all attributes of a class
//variables Class1 and Class2 must be bound
forall A do apply liftAttrsR(Class1,Class2,A);

//calling the rule for all possible matches in parallel
forall C1, C2, A do apply liftAttrsR(C1,C2,A);
//Apply a GT rule as long as possible for the entire model space
iterate choose C1, C2, A apply liftAttrsR(C1,C2,A)
```

Figure 3.8: Calling GT rules from ASM programs

Note the difference between the *as long as possible* and the *forall* execution modes: the former applies the rule once and only then does it select a next match as long as a match exists, while the latter collects all matches first, and then it applies the rule (one by one) for each of them in a single compound step.

3.4.3 Control Structure

To control the execution order and mode of graph transformations, VTCL includes language constructs that support the definition of complex control flow. As one of the main goals of the development of VTCL was to create a precise formal language, we included the basic set of Abstract State Machine (ASM) language constructs [25] that correspond to the constructs of conventional programming languages.

The basic elements of an ASM program are the rules (that are analogous with methods in OO languages), variables, and ASM functions. ASM functions are special mathematical functions, which store values in associative arrays (dictionaries). These values can be updated by ASM rules.

In VTCL, a special class of functions, called *native function*s, is also defined. Native functions are user-defined Java methods that can be called from the transformations. These methods can access any Java library (including database access, network functions, and so on), and also the VIATRA model space. This allows the implementation of complex calculations during the execution of model transformations.

ASMs provide complex model transformations with commonly used control structures in the form of built-in ASM rules, which are overviewed in Fig. 3.9. As a summary, ASMs provide

```
// variable definition
let X = 1 in ruleA
// variable (and ASM function) updates
update X = X + 1;
// print and log rules print a term to standard output or into the log
print("Print X: " + X + "\n");
log(info, "Log X: " + X);
// conditional branching by a logical condition or by pattern matching
if (X>1) ruleA else ruleB
if (find myPattern(X)) ruleA else ruleB
// exception handling: rule2 is executed only if rule1 fails
try rule1 else rule2
// calls the user defined ASM rule myRule with actual parameter X
call myRule(X)
// the sequencing operator: executes its subrules in the given order;
seq { rule1; rule2; }
// executes a non-deterministically selected rule from a set of rules
random { rule1; rule2; }
// iterative execution by applying rule1 as long as possible
iterate rule1;
//executes rule1 for a (non-deterministic) substitution of variable X
//which satisfies the pattern (or location) condition with X
choose X below M with (myAsmFun(X) > 0) do rule1
choose X below M with find myPattern(X) do rule1
//pseudo-parallel execution of rule1 for all substitution of variable X
//which satisfies the pattern (or location) condition with X
forall X below M with (myAsmFun(X) > 0) do rule1
forall X below M with find myPattern(X) do rule1
```

Figure 3.9: Overview of built-in ASM rules in VIATRA2

control structures including the sequencing operator (*seq*), rule calls to other ASM rules (*call*), variable declarations and updates (*let* and *update* constructs) and *if-then-else* structures, non-deterministically selected (*random*) and executed rules (*choose*), iterative execution (applying a rule as long as possible *iterate*), and the deterministic parallel rule application at all possible matches (locations) satisfying a condition (*forall*).

In addition to these core ASM rules, the VIATRA2 dialect of ASMs also includes built-in rules for manipulating the model space. As a result, elementary model transformation steps can be specified either in a declarative way (by graph transformation rules) or in an imperative way by built-in model manipulation rules. Main model manipulation rules are summarized in Fig. 3.10.

Most of these rules are rather straightforward, we only give more insight into the *copy* and *move* rules. The *copy* rule aims at copying an entity and all the recursive contents (i.e. the subtree of the entity) to a new parent. VIATRA2 provides two kinds of semantics for that copy operation: *keep_edges* and *drop_edges*. In the first case, all relations leading out from or into an entity placed anywhere below the copied entity are copied as well. In the latter case, only those relations are copied where both the source and the target entity are below the copied entity. In case of the *move* rule, an entity is moved (by force) to a new container by decoupling it from its old container. While this step may break the invariants of the old container, this problem is not as critical as in case of EMF as our constraints are checked at the end of the transformation.

These basic built-in ASM rules, combined with graph patterns and graph transformation rules, form an expressive, easy-to-use, yet mathematically precise language where the semantics of graph transformation rules can be given as ASM programs (see [J13]). The following example (in Fig. 3.11) demonstrates some of the main control structures.

```
// create a new entity C of type class and place it inside M
// the local name of C is automatically generated
new (class(C) in M);
// rename class C to "Product"
rename(C, "Product");
// create a new relation Attr of type attrs between C and A
// Attr is placed under its source C
new(attrs(Attr, C, A));
// Explicitly moves entity C (and all of its contents) to NewContainer
move(C, NewContainer);
// Retargets relation Attr to A1
setTo(Attr, A1);
// copy entity C and all of its contents directly under Container with
// ALL incoming and outgoing relations; the entity is accessed by CNew
copy(C, Container, CNew, keep_edges);
// copy entity C and all of its contents directly under Container but
// only copy relations between entities of the containment subtree of C
copy(C, Container, CNew, drop_edges);
// removes model element M together with its contents
delete(M);
```

Figure 3.10: Overview of model manipulation rules in VIATRA2

```
// An ASM rule is defined using the 'rule' keyword
rule main(in Model) =
 // Conversion from strings to model elements
 let M = ref(Model) in seq {
   //Print out some text
   print("The transformation of model " + M + " has started...\n");
   //Find all top-level classes below M and call rule printTopLevel
   forall Cl below M with find isTopClass(Cl) do
      call printTopLevel(Cl);
   //Apply a GT rule as long as possible for the entire model space
   iterate
          choose C1, C2, A apply liftAttrsR(C1,C2,A) do
             print("Attribute "+ name(A) + " is lifted from class " +
                   C1 + " to class " + C2 +"\n");
   //Write to log
   log(info,"Transformation terminated successfully.");
}
rule printTopLevel(in C) =
{
   print("Class " + name(C) + " is a top-level class.\n");
}
```

Figure 3.11: A sample ASM program driving a model transformation

3.5 Generic Transformations

3.5.1 An overview on generic transformations

Generic (or higher-order) transformations contain transformation rules where the types of certain objects are variables. An analogy can be made between general transformations and generic (or template) classes used in various object-oriented languages. However, while the parameters of generic classes are bound at design time, type variables in generic transformation rules are only substituted at transformation-time (run-time).

The advantages and drawbacks of higher-order transformations also show certain similarities with traditional logic frameworks. Higher-order logic is a very powerful description mechanism but it raises decidability (and performance) problems concerning automated reasoning

when compared with traditional first-order logic. Still, several powerful (higher-order) theorem provers have been applied for a large scale of practical verification problems.

Analogously, generic transformation rules offer a very high level of generality and compactness when compared to other (first-order) model transformation frameworks (especially, for higher-level transformations). A single generic rule can handle several situations where essentially the same rule pattern should be applied on objects of different types. On the other hand, the foundations of their type system (metamodeling framework) require some precautions to avoid certain well-known problems (see [14]). Furthermore, degradation in performance has been experienced in rewriting logic systems like Maude [35] when working with the meta-representations of large models.

To overcome these problems, we will build on VPM [J15], which is a dynamic metamodeling framework with fluid meta-levels. Moreover, generic transformation rules can also be turned into traditional first-order ones by meta-transformations [K25] to tackle performance problems.

3.5.2 Generic transformations in VIATRA2

To provide algorithm-level reuse for common transformation algorithms independent of a certain metamodel, VIATRA2 supports generic and meta-transformations, which are built on explicit instance-of relations of the VPM metamodeling framework. For instance, we may generalize rule *liftAttrsR* (of Fig. 3.7) as lifting something (e.g. an Attribute) one level up along a certain relation (e.g. parent). The generic example of Fig. 3.12 generalizes the previous GT rule (of Fig. 3.7) parameterized by types taken from arbitrary metamodels during execution time.

```
gtrule liftUp(inout CP, inout CS, inout A, inout ClsE, inout AttE,
              inout ParR, inout AttR) = {
   precondition pattern transCloseLHS(CP,CS,A, ClsE, AttE, ParR, AttR, Attr) =
   {
      // Pattern on the meta-level
      entity(ClsE);
      entity(AttE);
      relation(ParR,ClsE,ClsE);
      relation(AttR,ClsE,AttE);
      // Pattern on the model-level
      entity(CP);
      // Dynamic type checking
      instanceOf(CP,ClsE);
      entity(CS);
      instanceOf(CS,ClsE);
      entity(A);
      instanceOf(A,AttE);
      relation(Par,CS,CP);
      instanceOf(Par,ParR);
      relation(Attr,CS,A);
      instanceOf(Attr,AttR);
   }
   action  {
      delete (relation(Attr,CS,A));
      delete (instanceOf(Attr,AttR));
      new (relation(Attr2,CP,A))s;
      new (instanceOf(Attr2,AttR));
   }
}
```

Figure 3.12: Generic transformation rules

Compared to *liftAttrsR*, this generic rule *liftUp* has four additional input parameters: (i) *ClsE* for the type of the nodes containing the thing to be lifted (*Class* previously), (ii) *AttE* for the type

of nodes to be lifted (*Attribute* previously), and (iii) *ParR* (ex-*parent*) and (iv) *AttR* (ex-*attrs*) for the edge types.

When interpreting this generic pattern, the VIATRA2 engine first binds the type variables (*ClsE*, *ParR*, etc.) to types in the metamodel of a modeling language and then queries the instances of these types. Internally, this is carried out by treating subtype-of and instance-of relationships as special edges in the model space, which enables the easy generalization of traditional graph pattern matching algorithms.

Note that while other GT tools may also store metamodels and models uniformly in a common graph structure, only PROGRES [125] supports type parameters in rules, while none of them supports manipulation of (existing) type-instance relationship like the dynamic reclassification (retyping) of objects as in VIATRA2 .

In our solution, generic algorithms (e.g. transitive closure, graph traversals, fault modeling etc.) can be reused without changes in different metamodels. As a result, predefined general model transformation libraries can be assembled (like built-in libraries for traditional programming languages) to speed up transformation development.

As a conclusion, generic model transformation rules provide a compact and powerful specification mechanism to enhance reusability across model transformation problems.

3.6 FORMAL SEMANTICS OF GRAPH PATTERNS IN VTCL

As the main conceptual novelties of the VIATRA2 framework are related to the rich pattern language with recursive (and non-recursive) pattern calls, and arbitrary depth of negation, below we provide a formalization of the semantics of this crucial part. In order to avoid lengthy formal descriptions, we omit the formalization of other elements of the language, like ASM or GT rules. These already have a rich theoretical background, and VIATRA2 's contribution is less significant. For instance, a formal semantics of ASMs is given in [25], while an ASM formalization of GT rules is listed in [J13].

3.6.1 Formal representation of VPM models

A formal logic representation can be easily derived for VPM models. The vocabulary (signature) of VPM models can be derived directly from the Prolog-like syntax to include (i) *predicates* (i.e. boolean function symbols) *entity/1*, *relation/3*, *supertypeOf/2*, *instanceOf/2*, *in/2*, *below/2*, and (ii) traditional *boolean and arithmetic function symbols*. A *state of the model space* (denoted by $\mathcal{A}$) can be defined by an evaluation of these predicates and function symbols.

- *Entity.* A predicate *entity*(v) is evaluated to true in state $\mathcal{A}$ (denoted by $[\![\textit{entity}(v)]\!]^{\mathcal{A}}$), if an entity exists in the model space in state $\mathcal{A}$ uniquely identified by v. Otherwise the predicate is evaluated to false: $[\![\neg \textit{entity}(v)]\!]^{\mathcal{A}}$.

- *Relation.* A predicate *relation*(v, s, t) is true (denoted by $[\![\textit{relation}(v, s, t)]\!]^{\mathcal{A}}$), if the relation identified by v exists in the model space in state $\mathcal{A}$, and it leads from model element s to model element t. Otherwise, $[\![\neg \textit{relation}(v, s, t)]\!]^{\mathcal{A}}$.

- *SupertypeOf* A predicate *supertypeOf*(sup, sub) is evaluated as true (denoted by $[\![\textit{supertypeOf}(sup, sub)]\!]^{\mathcal{A}}$), if sup is a supertype of sub in the model space in state $\mathcal{A}$. Otherwise, $[\![\neg \textit{supertypeOf}(sup, sub)]\!]^{\mathcal{A}}$.

- *InstanceOf* A predicate *instanceOf*(ins, typ) is evaluated as true (denoted by $[\![\textit{instanceOf}(ins, typ)]\!]^{\mathcal{A}}$), if ins is an instance of typ in the model space in state $\mathcal{A}$. Otherwise, $[\![\neg \textit{instanceOf}(ins, typ)]\!]^{\mathcal{A}}$.

- *In* A predicate *in*(chi, par) is evaluated as true (denoted by $[\![\textit{in}(chi, par)]\!]^{\mathcal{A}}$), if chi is directly contained by par in the model space in state $\mathcal{A}$. Otherwise, $[\![\neg\textit{in}(chi, par)]\!]^{\mathcal{A}}$. Furthermore, we define predicate *below*(chi, anc) as the reflexive and transitive closure of in.

Entities predicates of the form type(id) are treated as a conjunction of three predicates while typed relations are handled accordingly: *entity*$(type) \wedge$ *entity*$(name) \wedge$ *instanceOf*$(id, type)$. Furthermore, we assume the existence of (an infinite pool) of fresh object identifiers, which will be assigned to newly created model elements.

3.6.2 Semantics of graph pattern matching

The formal semantics of graph patterns in VTCL will be defined as the set of all matches of a given pattern in a model space (see Table 3.1). For this purpose, VTCL patterns are first translated into a logic program (i.e. Prolog-like predicates). Then the semantics of this logic program will be defined as all the solutions which will be derived using standard relation database operations such as selection (σ), projection (π), inner join ($\bowtie$), and left outer join ($\ltimes$). Below we assume the reader's familiarity with these elementary relational operations.

Parameter passing. The semantics of a pattern is defined with respect to a given model space $\mathcal{A}$ and a set of input parameters passed to the pattern, which is encoded as a selection criterion $F_0 = \bigwedge_i X_i = v_i$.

Elementary predicates. First of all (Rows 1-3 in Table 3.1, the semantics of **elementary VPM predicates** (i.e. entity, relation, etc.) is simply the set of all corresponding tuples of the model space filtered by the selection σ fulfilling criterion F to restrict the result to the input parameters.

Simple patterns. Then a **simple pattern** (Row 4) is represented as a conjunction of VPM predicates. The semantics of a simple pattern is defined as the inner join ($\bowtie$) of matches retrieved by each predicate. In order to resolve name clashes of variables prior to the inner join operation, we rename all conflicting variable names by introducing new variables and extending the selection criterion F with corresponding equality constraints of (previously clashing) variables. For instance, if variable X is clashing, and thus it is renamed to variable Y, an equality constraint $X = Y$ is added to F.

Non-recursive calls. In case of **non-recursive pattern calls** (Row 5), we simply derive the inner join of each (non-recursive) pattern call by appropriate variable renaming as above.

Negative pattern calls. **Negative pattern calls** (Rows 6-7) are syntactically very close to ordinary pattern calls, however, their semantic treatment is essentially different. We assume that there are no recursive calls alternating between negative and positive patterns. That is, if the positive patterns transitively called by a pattern p are denoted by P while patterns transitively called by a negative pattern of p is denoted by N then $P \cup N = \emptyset$.

After that the left outer join ($\ltimes$) of the positive pattern $poscall$ and the negative patterns is calculated. Successful matches of the pattern are identified as rows where all non-shared variables of the negative patterns neg_j (i.e. non-shared with the positive pattern $poscall$) take the NULL value ($Y_{k,ns} = \varepsilon$).

	VTCL grammar (simplified)	Derived predicates	Semantics
1	$fact$ = ***entity***(V) (or ***relation***(V,S,T))	$fact(V) \leftarrow$ ***entity***(V) $fact(V,S,T) \leftarrow$ ***relation***(V,S,T)	$[\![fact(V)]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_F(\{v \mid [\![$***entity***$(v)]\!]^{\mathcal{A}}\})$ $[\![fact(V,S,T)]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_F(\{(v,s,t) \mid [\![$***relation***$(v,s,t)]\!]^{\mathcal{A}}\})$
2	$fact$ = ***supertypeOf***(A,B) (or ***instanceOf***(A,B))	$fact(A,B)$ $\leftarrow$ ***supertypeOf***(A,B) (or ***instanceOf***(A,B))	$[\![fact(A,B)]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_F(\{(a,b) \mid [\![$***supertypeOf***$(a,b)]\!]^{\mathcal{A}}\})$ (or ***instanceOf***(a,b))
3	$fact$ = ***in***(A,B) (or ***below***(A,B))	$fact(A,B) \leftarrow$ ***in***(A,B) (or ***below***(A,B))	$[\![fact(A,B)]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_F(\{(a,b) \mid [\![$***in***$(a,b)]\!]^{\mathcal{A}}\})$ (or ***below***(a,b))
4	$simple = fact_1$**;** $\ldots fact_n$**;**	$simple(\overline{X}) \leftarrow$ $fact_1(\overline{X_1}) \wedge \cdots \wedge fact_1(\overline{X_n})$	$[\![simple(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} \overset{F_1}{\bowtie}_l([\![fact_l(\overline{Y_l})]\!]_{F_1}^{\mathcal{A}})$ where $F_1 = F \wedge \bigwedge_k X_{i,k} = Y_{j,k}$ for each variable renaming.
5	$poscall$ = $simple$ **find (** $base_1$ **);** $\ldots$ **find (** $base_n$ **);**	$poscall(\overline{X}) \leftarrow$ $simple(X_0) \wedge \bigwedge_j base_j(\overline{X_j})$	$[\![poscall(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![simple(\overline{Y_0})]\!]_{F_1}^{\mathcal{A}} \overset{F_1}{\bowtie} [\![base_j(\overline{Y_j})]\!]_{F_1}^{\mathcal{A}}$ with $F_1 = F \wedge \bigwedge_k (X_k = Y_k)$ for each var renaming.
6	neg = **neg find (** $\{base \mid recur\}$ **);**	$neg(\overline{X}) \leftarrow base(\overline{X})$	$[\![neg(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![base(\overline{X})]\!]_F^{\mathcal{A}}$
7	$base = poscall\ neg_1, \ldots, neg_n$	$base(\overline{X}) \leftarrow poscall(\overline{X_0}) \wedge \bigwedge_j(\neg neg_j(\overline{X_j}))$	$[\![base(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_{F_2}([\![poscall(\overline{Y_0})]\!]_{F_1}^{\mathcal{A}} \overset{F_1}{\ltimes} ([\![neg_j(\overline{Y_j})]\!]_{F_1}^{\mathcal{A}}))$ where $F_1 = F \wedge \bigwedge_k X_k = Y_k$ for each variable renaming, and $F_2 = F_1 \wedge \bigwedge Y_{k,ns} = \varepsilon$ for non-shared variables of patterns neg_j and $poscall$.
8	$recur = base$ **find (** $recur_1$ **);** $\ldots$ **find (** $recur_n$ **);**	$recur(\overline{X}) \leftarrow base(\overline{X_0}) \wedge recur_1(\overline{X_1}) \wedge \cdots \wedge recur_n(\overline{X_n})$	$[\![recur(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} lfp([\![base(\overline{Y_0})]\!]_{F_1}^{\mathcal{A}} \overset{F_1}{\bowtie} [\![recur_1(\overline{Y_1})]\!]_{F_1}^{\mathcal{A}} \overset{F_1}{\bowtie} \ldots \overset{F_1}{\bowtie} [\![recur_n(\overline{Y_n})]\!]_{F_1}^{\mathcal{A}})$ where $F_1 = F \wedge \bigwedge_k X_k = Y_k$ for each variable renaming
9	$body = recur$ (**check** $cond$)?	$body(\overline{X}) \leftarrow recur(\overline{X}) \wedge check(\overline{X})$	$[\![body(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![recur(\overline{X})]\!]_{F_1}^{\mathcal{A}}$ where $F_1 = F \wedge check(\overline{X})$
10	**pattern** $patt$ = **{** bod_1 **} or {** bod_2 **}**	$patt(\overline{X}) \leftarrow bod_1(\overline{X}) \vee bod_2(\overline{X})$	$[\![patt(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![bod_1(\overline{X})]\!]_F^{\mathcal{A}} \cup [\![bod_2(\overline{X})]\!]_F^{\mathcal{A}}$

Table 3.1: Deriving predicates for VTCL patterns

Recursive pattern calls. In VTCL, arbitrary recursive pattern calls are allowed (as long as recursion and negation are not alternating), which is a rich functionality. Obviously, we require the presence of a **base pattern** which is a simple pattern extended with non-recursive calls and negative patterns. The semantics of recursive calls is defined in a bottom up way as the least fix point of the inner join operation $[\![base]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} [\![recur_1]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} \ldots \overset{F_1}{\bowtie} [\![recur_n]\!]_F^{\mathcal{A}})$. Initially, $[\![recur_i^{(0)}]\!]_F^{\mathcal{A}} = \emptyset$ for all i. Then in the first iteration, it collects all matches derived by the base cases of **recursive patterns**, i.e. $[\![recur_i^{(1)}]\!]_F^{\mathcal{A}} = [\![base_i]\!]_F^{\mathcal{A}}$ (i.e. its own base case). In all upcoming iterations, $[\![recur^(i+1)]\!]_F^{\mathcal{A}}$ is defined as $[\![base]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} [\![recur_1^{(i)}]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} \ldots \overset{F_1}{\bowtie} [\![recur_n^{(i)}]\!]_F^{\mathcal{A}}$ with the appropriate variable renaming. This step is executed until a fixpoint is reached, which might cause non-termination for ill-formed programs.

Check condition and OR patterns. **Check conditions** (Row 9) in a pattern simply extend the selection criteria F, while the result of OR-patterns (Row 10) is defined as the union of the results for each pattern body.

3.6.3 Semantics of calling graph patterns from ASMs

In case of model transformations, graph patterns are called from ASM programs by (i) supplying input parameters, and (ii) defining whether pattern matching should be initiated in *choose* or *forall* mode by quantifying free variables of the pattern. Note that the same pattern can be called with different variable binding, e.g., a pattern parameter can be an input parameter at a place, while it can be quantified by *forall* at a different location.

When using the *choose* construct to initiate pattern matching, the free variables will be substituted by **existential quantification**, i.e. only one (non-deterministically selected) match will be retrieved from the result set of the pattern.

However, if the pattern is called using the *forall* construct, this means **universal quantification** for the pattern parameters (head variables), thus all possible values of the head variables that satisfy the pattern will be retrieved, and the ASM body of the *forall* rule will be executed on each pattern one by one.

Finally, patterns may also contain internal variables which appear in the body but not in the formal parameter list (head), which variables are quantified existentially. Note that universally quantified variables take precedence over existentially quantified ones, i.e. we try to find one substitution of existentially quantified variables for each valid substitution of universally quantified ones.

Formalization. In order to formalize this behavior, let $\overline{X} = \overline{X^{in}} \cup \overline{X^f} \cup \overline{X^b}$ denote all the variables in the body of a pattern $patt$ where $\overline{X^{in}}$ is supplied as input parameters (with assignments $X_i^{in} = v_i$ that constitute the initial filtering condition F), $\overline{X^f}$ denote the free variables in the pattern head, while $\overline{X^b}$ denote the variables appearing only in the pattern body.

Then the semantics (i.e. result set) retrieved by each construct is defined by projecting the result set to the columns of pattern variables only. As a consequence, variables of the body $\overline{X^b}$ are always evaluated in an existential way, no matter how many matches they have.

- $[\![$**choose** $\overline{X^f}$**with find** $patt(\overline{X^{in}} \cup \overline{X^f})]\!]_F^{\mathcal{A}} \overset{def}{=}$ any $v \in \pi_{\overline{X^{in}} \cup \overline{X^f}}([\![patt(\overline{X})]\!]_F^{\mathcal{A}})$, i.e. *any value* in the result set projected to columns of the pattern variables only. If $[\![patt(\overline{X})]\!]_F^{\mathcal{A}} = \emptyset$, then the choose construct becomes inconsistent to cause backtracking in ASM.

- $[\![$**forall** $\overline{X^f}$**with find** $patt(\overline{X^{in}} \cup \overline{X^f})]\!]_F^{\mathcal{A}} \overset{def}{=} \pi_{\overline{X^{in}} \cup \overline{X^f}}([\![patt(\overline{X})]\!]_F^{\mathcal{A}})$, i.e. the (possibly empty) result set projected to the columns of pattern variables only.

Finally, note that while a *forall* rule collects all the matches for a pattern in a single step, its body is executed sequentially one by one on the matches. For the moment, only partial checks are carried out during run-time to detect conflicts in the execution of different bodies in a *forall* rule.

3.7 Related Work

Since VIATRA2 can also be regarded as a graph transformation tool, we now compare the advanced language constructs to similar constructs available in the most popular graph transformation tools, namely, AGG [54], ATOM3 [42], FUJABA [103], GReAT [16], PROGRES [125], VMTS [90] and VIATRA2 citeVIATRA2. In addition, we also include ATL [77] in our comparison as an advanced model transformation language that is not built on graph transformation principles.

More specifically, we compare in Fig. 3.13 the advanced constructs for specifying *patterns* (such as negative application conditions, multi-objects, path expressions, constraints), *control structures* (such as iterative and parallel rule applications, or parameter passing between rules), and general *transformation features* (bidirectional transformations, template based code generation or generic transformations). Obviously, the corresponding ATL constructs are only approximately "equivalent" due to the differences between the graph-centric and object-centric paradigms of these transformation languages.

		ATL	AGG	ATOM3	FUJABA	GReAT	PROGRES	VMTS	VIATRA2
Patterns	Negative	in OCL constraints	+	+	+	+	+	X	arbitrary level
	Multi-object	simulated by forall() in OCL constraints	X	X	+	node multipl.	+	node multipl.	simulated by forall quantif. of node variables
	Recursiveness	recursive helpers	X	X	path expr	X	path expr	X	recursive patterns
	Constraint	OCL	graph + Java	Python	Java	OCL	+	OCL	graph patterns
Control structure	Parallel	by default	X	X	simulated by multi-objects	by default	+	X	+
	Iterate	foreach	simulated by layers	simulated by priorities	+	+	+	+	+
	Param pass	+	X	X	+	+	+	+	+
Xform	Bidirectional	X	X	TGG plugin	TGG plugin	X	X	X	X
	Code template	X	X	X	+	X	X	X	+
	Generic Xform	X	X	X	X	X	node types	X	+

+ = Supported X = Not available

Figure 3.13: Comparison of advanced language constructs in model transformation tools

We would like to point out that VIATRA2 provides a more general support than any of these graph transformation tools in the following areas:

- Pattern calls in VIATRA2 facilitate the *reusability of elementary patterns* to construct complex patterns and rules. Fully declarative pattern calls in both LHS and RHS of GT rules are a novel feature compared to other GT tools. In practice, it turns out to be a very powerful technique to decompose complex model transformations into reusable pieces.
- VIATRA2 allows *negative conditions with arbitrary depth of negation*. This way, negative patterns are not allowed to have negative patterns in turn. Other GT tools offer negative conditions with a single depth of negation.
- Graph patterns in VTCL allow for *full recursion*. The best existing solution for recursion (used in PROGRES and FUJABA) is called path expressions, which are basically regular

expressions over edges. This case structural recursion appears only on a single path of edges.

- Only the transformation language of VIATRA2 supports *generic transformations* (transformation rules having metamodel-level type parameters and dynamic retyping of model elements) and *meta transformations* (rules manipulating other rules). This allows to create highly reusable libraries for generic algorithms which are applicable to different metamodels.

On the other hand, bidirectional transformations (provided by triple graph grammars [124] in FUJABA and ATOM3) are not supported in VIATRA2 . For all other language features, the VIATRA2 solution is (at least) comparable to some existing approaches available in graph transformation tools. Finally, the approach of integrating graph transformation and ASMs is quite novel.

For other related model transformation tools, it is worth mentioning MT [134], which uses a powerful pattern language with multiplicities for pattern objects. On the one hand, such multiplicities in MT provide a finer control over matching of a single object, i.e. to have exactly five matches for an element. On the other hand, recursive patterns in VTCL provide more general recursion.

Similar structuring concepts (like rules, patterns templates) are proposed in Tefkat [87]. On the one hand, only simple recursion is supported, and generic patterns are out of scope for Tefkat. On the other hand, the extend and supersede constructs in Tefkat are powerful mechanisms for reusability.

3.8 CONCLUSIONS

We presented the model transformation language of the VIATRA2 framework, which provides a rule and pattern-based transformation language for manipulating graph models by combining graph transformation and abstract state machines into a single paradigm. In addition, powerful language constructs are provided for multi-level metamodeling to design modeling languages.

After a comparison with the transformation language of leading graph transformation tools, we can conclude that the transformation language of VIATRA2 offers advanced constructs for querying (e.g. recursive graph patterns) and manipulating models (e.g. generic and meta-transformation rules) in unidirectional transformations which are frequently used in formal model analysis to carry out abstractions.

Chapter 4

Model Transformation by Example

4.1 Introduction

The efficient design of automated model transformations between modeling languages has become a major challenge to model-driven engineering (MDE) by now. Many highly expressive transformation languages and efficient model transformation tools are emerging to support this problem. The evolution trend of model transformation languages is characterized by gradually increasing the abstraction level of such languages to declarative, rule-based formalisms as promoted by the QVT (Queries, Views and Transformations) [108] standard of the OMG.

However, a common deficiency of all these languages and tools is that their transformation language is substantially different from the source and target models they transform. As a consequence, transformation designers need to understand not only the transformation problem, i.e. how to map source models to target models, but significant knowledge is required in the transformation language itself to formalize the solution. Unfortunately, many domain experts, who are specialized in the source and target languages, lack such skills in underlying transformation technologies.

Model transformation by example (MTBE) is a novel approach (first introduced in [K22]) to bridge this conceptual gap in transformation design. The essence of the approach is to derive model transformation rules from an initial prototypical set of interrelated source and target models, which describe critical cases of the model transformation problem in a purely declarative way. A main advantage of the approach is that transformation designers use the concepts of the source and target modeling languages for the specification of the transformation, while the implementation, i.e. the actual model transformation rules are generated (semi-)automatically. In our context, (semi-)automatic rule generation means that transformation designers give hints how source and target models can *potentially* be interconnected in the form of a mapping metamodel. Then the actual contextual conditions used in the transformation rules are derived automatically based upon the prototypical source and target model pairs.

The current chapter proposes the use of inductive logic programming [101] (ILP) to automate the model transformation by example approach. ILP can be defined as an intersection of inductive learning and logic programming as it aims at the inductive construction of first-order clausal theories from examples and background knowledge, thus using induction instead of deduction as the basic mode of inference. As the main practical advantage of this interpretation, we demonstrate that by using existing ILP tools, we can achieve a high level of automation for MTBE using relatively small examples.

The rest of this chapter (which is primarily based on [J1]) is structured as follows. Sec-

tion 4.2 gives an overview of the core concepts of the MTBE approach. Section 4.3 summarizes the inputs required for MTBE. In Section 4.4, which is the central part of the current chapter, we give a detailed presentation on how to automate MTBE using inductive logic programming. In Section 4.5, we collected some known limitations of our approach, while Sec. 4.6 sketches a prototypical tool chain that we used for carrying out our case studies. Finally, Section 4.7 discusses related work and Section 5.2.5 concludes our chapter.

4.2 Model Transformation by Example (MTBE)

4.2.1 Overview of Model Transformation By Example

Model transformations by example (MTBE) [K22] is defined as a highly iterative and interactive process as illustrated in Fig. 4.1.

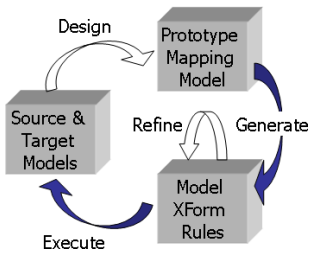


Figure 4.1: Model Transformation By Example: Process Overview

Step 1: Set-up of prototype mapping models. The transformation designer assembles an initial set of interrelated source and target model pairs, which are called *prototype mapping models* in the rest of the chapter. These prototype mapping models typically capture critical situations of the transformation problem by showing how the source and target model elements should be interrelated by appropriate mapping constructs.

Step 2: Automated derivation of rules. Based on the prototype mapping models, the MTBE framework should synthesize a set of model transformation rules, which correctly transform as many prototypical source models into their target equivalents as possible.

Step 3: Manual refinement of rules. The transformation designer can refine the rules manually at any time by adding attribute conditions or providing generalizations of existing rules.

Step 4: Automated execution of rules. The transformation designer validates the correctness of the synthesized rules by executing them on additional source-target model pairs as test cases, which will serve as additional prototype mapping models. Then the development process is started all over again.

The main vision of the "model transformations by example" approach is that the transformation designer mainly uses the concepts of the source and target languages as the "transformation language", which is very intuitive. He or she does not need to learn a new formalism for capturing model transformations.

While the current chapter focuses on automating the "model transformation by example" approach, we still regard MTBE as a highly iterative and interactive process. Our experience also shows that it is very rare that the *final* set of transformation rules is derived right from the *initial*

set of prototype models. Furthermore, transformation designer can overrule the automatically generated rules at any time, especially, when certain critical abstractions or generalizations are not detected automatically.

Concerning correctness issues, one would expect as a minimum requirement that the derived model transformation rules should correctly transform all prototypical source models into their target equivalent. However, this is not always practical, since overspecification or incorrect specification in prototype mapping models may decrease the chance of deriving a meaningful set of model transformation rules. Since MTBE takes prototype mapping models as specifications, (unintended) omissions in them might easily result in incorrect rules. Therefore, MTBE approaches should ideally tolerate a certain amount of "noise" when processing prototype mapping models.

4.2.2 Steps of automation

From a technical point of view, the process of model transformation by example can be split into the following phases to support the semi-automatic generation of transformation rules:

1. *Set up an initial prototype mapping model.* In the first step, an initial prototype mapping model is set up manually from scratch or by using existing source and target models.

2. *Context analysis.* Then we identify (positive and negative) constraints in the source and target models for the presence of each mapping node. For instance, only top-level classes are related to database tables or a table related to a class always contains a primary key column. For this purpose, we first examine the contexts of all mapped source and target nodes.

3. *Connectivity analysis.* For each edge in the target metamodel, we identify contextual conditions (in the source and mapping models) for the existence of that target edge.

4. *Derive transformation rules for target nodes.* Then we derive transformation rules for all (types of) mapping nodes that derive only target nodes using the information derived during context analysis. Informally, the context of source nodes will identify the precondition of the derived model transformation rules, while the context of target nodes will define the postcondition of model transformation rules. As a result, we create target nodes from source nodes interconnected by a mapping structure (of some type).

5. *Derive transformation rules for target edges.* Finally, we derive transformation rules for each target edge based upon the information gained during connectivity analysis of source and target elements.

6. *Iterative refinement.* The derived rules can be refined at any time by extending the prototype mapping model or manually generalizing the automatically generated rules.

In the current chapter, we discuss how the MTBE approach can be automated by using inductive logic programming [101] as an underlying framework. Our goal is to show that it is possible to construct relatively small prototype mapping models for practical problems from which the complete set of model transformation rules can be derived semi-automatically. Furthermore, we also identify critical transformation problems where our approach failed to derive a complete solution.

4.3 INPUTS OF MODEL TRANSFORMATION BY EXAMPLE

Model transformation by example takes a set of prototypical interconnected source and target model pairs as inputs. These interconnections are described by mapping metamodels (Sec. 2.1.3) and prototype mapping models (Sec. 4.3.1), which are prototypical instances of the mapping metamodel to capture semantic cornercases of the model transformation problem as pairs of (interconnected) models. We also collect our assumptions for the structure of prototype mapping models that we rely upon for automation (Sec. 4.3.2). The specificities of our approach will be demonstrated using the object-relational mapping as a running example (Sec. 2.2.3.

Assumptions on mapping metamodel. It is worth pointing out that we assume that transformation designers provide the mapping metamodel as input to model transformation by example. This metamodel already contains certain hints from the transformation designer on the actual transformation rules. For instance, we learn from the metamodel that classes are expected to be transformed to tables and columns.

However, the mapping metamodel does not specify (i) under what condition a class is transformed into a table, (ii) if the same class can be mapped to different tables, (iii) if two classes are mapped into the same tables, or (iv) if classes are ever transformed to columns at all. These contextual conditions will be defined below implicitly by prototype mapping models. The automation discussed in this chapter will exclusively focus on automating the derivation of model transformation rules to precisely capture contextual conditions with respect to its inputs, namely, the mapping metamodel and prototype mapping models.

Finally, we regard the derivation of mapping metamodel as a separate automation problem, which is subject to future work. First results for (semi-)automating this step were reported in [47].

4.3.1 Prototype mapping models

Prototype mapping models can be obtained by interrelating any pair of existing (real) source and target models. However, prototype mapping models are preferably small, thus they are rather created by hand to incorporate critical situations of the transformation problem. These prototype mapping models can also serve as test cases later on.

Example 4.1 A sample prototype mapping model is depicted in Fig. 4.2 for illustration purposes. This prototype mapping model contains three UML classes (*Employee*, *Manager* and *Clerk* where the first is the *parent* of the other two), an attribute *boss* of type *Manager* belonging to class *Clerk* and an attribute *favourite* of type *Clerk* belonging to the class *Manager*.

The prototype mapping model captures that two relational database tables are present in the target model (*tMngr* and *tClerk*). Both tables have three columns:

- The columns of *tMngr*: *idMngr* for the unique identifier, *kindMngr* for storing the kind of instance, and *favouriteClerk* which is the target equivalent of the *favourite* attribute together with a foreign key *fkeyFavourClerk*
- The columns of *tClerk*: *idClerk*, *kindClerk* and *clerkBoss*, where the latter is the target equivalent of the *boss* attribute together with a corresponding foreign key *fkeyClerkBoss*.

It is worth pointing out that not all source nodes are necessarily linked to a mapping node. In such a case, the corresponding source element does not have an equivalent in the target model (see the prototype mapping model of Fig. 4.3 for an example). However, each target element is required to be linked to exactly one mapping node, otherwise, their existence is not causally dependent on the source model, which will be one of our assumptions.

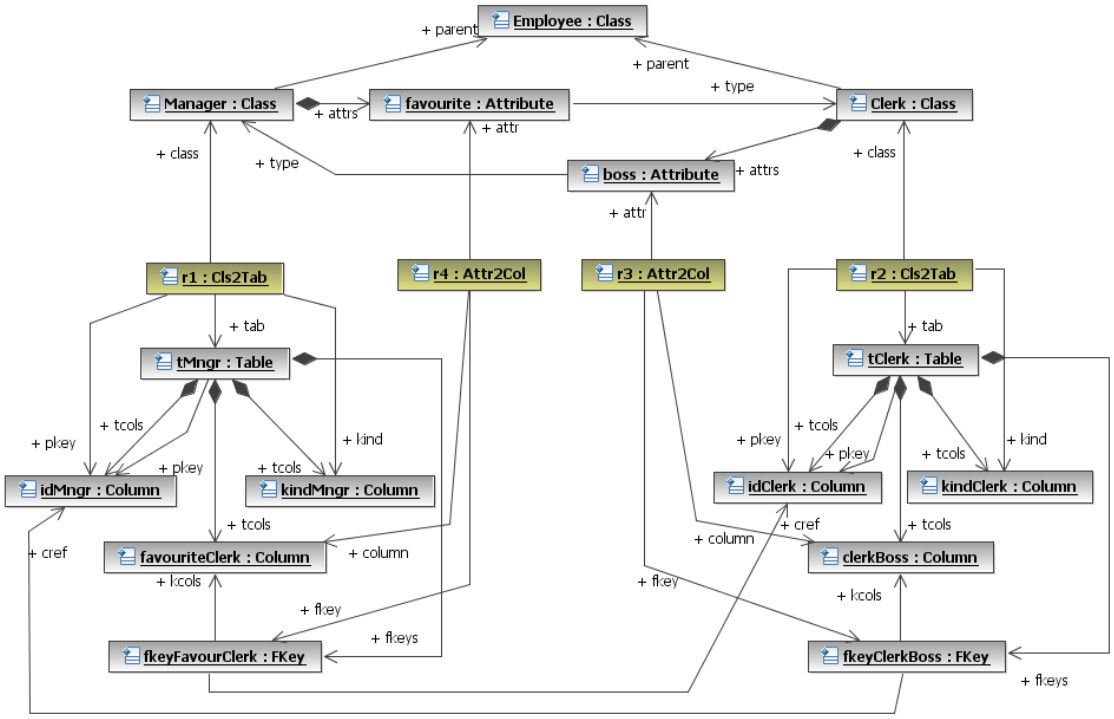


Figure 4.2: A prototype mapping model

4.3.2 Assumptions

For this derivation process we make the following assumptions for the structure of the prototype mapping models:

Assumption 1: Each mapping node is connected to all the source nodes on which it is causally dependent. This requirement guarantees that prototype mapping models describe exactly the intent of the transformation designers, thus they can be used as a specification for the automation step.

Assumption 2: Each mapping node is connected to all the target nodes which cannot exist without each other (i.e. belong to the same component). For instance, in our mapping, a table generated from a class always has a primary key column, thus both are linked to the same mapping node. Note that this is a major syntactic difference compared to [K22], however, it fully corresponds to the idea of weaving models [22] where complex mapping structures are used.

Assumption 3: Each target node is linked to exactly one mapping node. For instance, a table and its primary key column will be linked to the same mapping node. This requirement prescribes that the transformation is deterministic in a sense that the creation of each target node is uniquely identified by a mapping node and a corresponding mapping edge, and no merging of target nodes is required.

Assumption 4: The existence of a node in the target model depends only on the existence of a certain mapping node, i.e. source and target models are dependent on each other only indirectly via mapping structures.

Assumption 5: The existence of an edge in the target model depends only on contextual conditions of the source model and the existence of certain structure (but does not directly depend on the target model itself).

While these assumptions seem to cover a large set of practical model transformations, we will also discuss certain transformation problems in Sec. 4.5 where some of these conditions do not hold, and we ran into problems when automating MTBE.

4.4 Automating Model Transformation by Example

We now discuss how inductive logic programming can be used to automate the model transformation by example approach. First we sketch how prototype mapping models can be constructed and mapped into corresponding Prolog clauses (Sec. 4.4.1). Then, we discuss one by one how to automate each phase of MTBE (Sec. 4.4.2-4.4.5).

4.4.1 From prototype mapping models to Prolog clauses

Source and target models are mapped into corresponding Prolog clauses following a straightforward representation where each node type in the source or target metamodel is mapped into a unary predicate, and each edge type in the metamodel is transformed into a binary predicate. In order to avoid name clashes, the name of the source node type is generated as a prefix for binary predicates for edges.

- On the one hand, (source or target) model nodes correspond to a ground predicate (over the unique identifier which represents the object), which predicate defines the type of that node.
- On the other hand, (source or target) edges have no unique identifiers in our representation, the corresponding binary predicate defines the source and target nodes, respectively.

Note that this is not a conceptual limitation of our approach: this mapping to Prolog clauses could be easily refined to incorporate identifiers of edges. Unsurprisingly, there is a penalty in the performance of the underlying ILP engine to incorporate this change.

Example 4.2 For instance, in the UML part of the prototype mapping model of Fig. 4.2, a class *Clerk* with an attribute *boss* of type class *Manager* can be represented by the following Prolog clauses (provided that *clerk*, *manager* and *boss* are unique identifiers this time).

```
% Clauses for source model
class(clerk).
class(manager).
attribute(boss).
attribute(favourite).
class_attrs(manager,favourite).
class_attrs(clerk,boss).
attribute_type(boss,manager).
attribute_type(favourite,clerk).
```

The representation of its corresponding target model (see Fig. 4.2) is the following.

```
% Clauses for target model
table(tMngr).
column(idMngr).
column(kindMngr).
column(favouriteClerk).
table_tcols(tMngr,idMngr).
table_tcols(tMngr,kindMngr).
```

```
table_tcols(tMngr,favouriteClerk).
table_pkey(tMngr,idMngr).
fkey(fkeyFavourClerk).
table_fkey(tMngr,fkeyFavourClerk).
fkey_cref(fkeyFavourClerk,idClerk).
fkey_kcols(fkeyFavourClerk,favouriteClerk).

table(tClerk).
column(idClerk).
column(kindClerk).
column(clerkBoss).
table_tcols(tClerk,idClerk).
table_tcols(tClerk,kindClerk).
table_tcols(tClerk,clerkBoss).
table_pkey(tClerk,idClerk).
fkey(fkeyClerkBoss).
table_fkey(tClerk,fkeyClerkBoss).
fkey_cref(fkeyClerkBoss,idMngr).
fkey_kcols(fkeyClerkBoss,clerkBoss).
```

Representation of mapping models

Mapping (weaving) models are represented in a slightly different way in order to improve the performance of the ILP engine. In Sec. 2.1.3, we made an assumption on the structure of mapping models, namely, that each mapping node and its outgoing edges can be represented by a tuple

$$ref(r_i, src_1, \ldots, src_n, trg_1, \ldots, trg_m).$$

As a consequence a straightforward representation of mapping models is the following:

```
% Clauses for mapping model
cls2tab(r1,manager,tMngr,idMngr,kindMngr).
cls2tab(r2,clerk,tClerk,idClerk,kindClerk).
attr2col(r3,boss,clerkBoss,fkeyClerkBoss).
attr2col(r4,favourite,favouriteClerk,fkeyFavourClerk).
```

Furthermore, each mapping node is uniquely identified by (ordered) source nodes and (ordered) target nodes ($src_1, \ldots, src_n$ and $trg_1, \ldots, trg_m$, respectively). Therefore, the identifier r_i can be omitted from the tuple, moreover, the tuple can be projected to the source and target elements without changing its truth value, i.e.

$$\begin{aligned} ref(r_i, src_1, \ldots, src_n, trg_1, \ldots, trg_m) &\iff \\ ref(src_1, \ldots, src_n) &\iff \\ ref(trg_1, \ldots, trg_m). & \end{aligned}$$

As a consequence, we can interchangeably use the following clauses in the different phases of MTBE:

```
% Alternative clauses for mapping models
cls2tab(r1,manager,tMngr,idMngr,kindMngr).
cls2tab(manager).
cls2tab(tMngr,idMngr,kindMngr).
```

Inheritance and helper edges

While the simplified metamodels of Fig. 2.2 do not contain generalization, we emphasize that the inheritance hierarchy of metamodel nodes and edges (i.e. the generalization of classes and the refinement of association ends as in MOF 2.0) can be mapped into Prolog clauses of the form

superclass(X) :- subclass(X)

when *superclass* is a generalization of *subclass* in some metamodel. Such clauses will be part of our background knowledge.

Various model transformation approaches introduce the concept of helper (derived) edges in order to reduce the complexity of individual transformation rules. When the actual model transformation rules are derived, we assume that such helper edges of the source language are already part of our background knowledge. As a result, helper edges enable the ILP engine to derive more general hypothesis as an output.

Background knowledge on helper edges can be obtained in two different ways.

- We allow domain experts to directly add helper information to the knowledge base.
- However, we can also use ILP to derive such helper information automatically in a preprocessing phase carried out on the source and target languages separately.

Example 4.3 In our running example, the background knowledge may contain the following definitions defined by some domain expert (currently on the Prolog level).

```
ancestor(X,Y) :- class_parent(X,Y).
ancestor(X,Y) :- class_parent(Z,Y), ancestor(X,Z).
```

This definition of the *ancestor* relation can also be taught in a preprocessing phase by the ILP engine with an appropriate set of positive and negative examples (i.e. using the same technique as described below). In such a case, the domain expert only gives a hint that the *ancestor* relation may be important (by adding it to the metamodel as a derived element), and supplies an appropriate set of examples. In the sequel, we assume the presence of such helper relations in the background knowledge, which were derived by analyzing the source and target models prior to the model transformation itself.

Moreover, by using abductive learning [6, 100] techniques, Aleph can also synthesize the *ancestor* relation on demand, i.e. when learning other predicates. In this case, we only need to assume that our background knowledge contains sound but not necessarily complete information on helper edges like *ancestor*, i.e. the domain expert needs to give some positive examples for the *ancestor* edge.

Furthermore, there is intensive research on predicate learning [44] in the field of inductive logic programming when absolutely no hint is required from domain experts on helper edges, i.e. not even the existence of *ancestor* edge is required to be given as a hint. However, Aleph unfortunately does not yet support this feature.

4.4.2 Context analysis

In this phase, we identify constraints in the source and target models for the presence of each mapping node.

Context analysis for the source model

In case of context analysis of the source model, our background knowledge B will consist of all the facts derived from the source model (by taking the source projection of the prototype mapping model), and the positive and negative facts (E^+ and E^-) will consist of tuples on the mapping node in question. This way, a separate ILP problem will be derived for each mapping node.

When deriving negative facts from a prototype mapping model, we currently build on closed world assumption as a frame condition, i.e. negative facts are derived for all identifiers part of the source model and not listed in positive facts. However, this is not the only possibility, and

one can explicitly ask the user to provide (an incomplete) list of negative examples. While this latter approach seems to be more cumbersome, in many case the ILP engine can better tolerate noise, i.e. when transformation designers unintendedly include (or exclude) certain elements from the prototype mapping model.

Example 4.4 For instance, we will identify contextual conditions in the source model when a mapping node of type *attr2col* should appear. For this purpose, we construct B from Fig. 4.2 to obtain the facts listed above. For positive facts, we have *attr2col(boss)*, and *attr2col(favourite)* and for negative facts, we can state *attr2col(manager)*, *attr2col(employee)* and *attr2col(clerk)*. In Aleph, all these specifications need to be listed in separate files, but for presentation purposes, we will list them together.

```
% Background knowledge
class(clerk).
class_attrs(clerk,boss).
attribute(boss).
class(manager).
class_attrs(manager,favourite).
attribute_type(boss,manager).
attribute_type(favourite,clerk).
% Positive facts
attr2col(boss).
attr2col(favourite).
% Negative facts
attr2col(clerk).
attr2col(manager).
attr2col(employee).
```

Aleph will automatically derive the following rule as hypothesis by using core inductive logic programming algorithms:

```
attr2col(X) :- attribute(X).
```

Note that this result fulfills our expectations provided that only well-formed models are considered when language-specific constraints are checked separately.

However, we might want to specify that attributes are required to be attached to classes, and that each attribute is required to have a type in order to be transformed into a corresponding column. Since ILP derives the most general solution, these constraints are not incorporated in the solution by default. Therefore, we enrich our background knowledge and negative facts with fictitious model elements.

Example 4.5 Let us revisit the previous example by enriching background knowledge

```
% Background knowledge
% ... as before +
attribute(notype).
attribute(noattrs).
class_attrs(notype,manager).
class_attrs(noattribute,clerk).
attribute_type(noattrs,manager).
attrs(noattribute,manager).
% Negative facts
% ... as before
% Negative facts
% ... as before +
attr2col(noattrs).
attr2col(notype).
attr2col(noattribute).
```

As a result, the ILP engine will derive the following rule:

```
attr2col(A) :- attribute(A), class_attrs(B,A),
               attribute_type(A,C).
```

This rule will properly handle incomplete model as well. The price we paid for that is that both the background knowledge B and the negative facts E^- need to be extended with carefully selected cases, which can be cumbersome. Fortunately, if a sufficient number of real source and target model pairs are available, they might already cover cases to handle such incomplete models. Anyhow, it is a subject of future research to incorporate language constraints into automatically generated transformation rules.

In the general case, the ILP engine might derive multiple rules as a hypothesis, which means a disjunction of the different bodies. This is normal for the context analysis of the source model, since the same type of mapping nodes can be used with different source contexts.

Context analysis for the target model

As for the context analysis of the target model, the ILP engine needs to identify trivial hypotheses due to Assumption 3 (discussed in Sec. 4.3.2), which prescribes causal dependency of target nodes on the mapping structure. Therefore, we reverse the direction of our investigations, and use predicates corresponding to mapping structures as background knowledge, predicates corresponding to the nodes of the target model as positive and negative examples.

Example 4.6 Below we present an example for identifying the target context for *FKey*s.

```
% Background knowledge
attr2col(favouriteClerk,fkeyFavourClerk).
attr2col(clerkBoss,fkeyClerkBoss).
cls2tab(tMngr,idMngr,kindMngr).
cls2tab(tClerk,idClerk,kindClerk).
% Positive facts
fkey(fkeyClerkBoss).
% Negative facts
fkey(clerkBoss).
fkey(tMngr).
fkey(idMngr).
fkey(kindMngr).
fkey(tClerk).
fkey(idClerk).
fkey(kindClerk).
```

As a result, the ILP engine will derive the following hypothesis for the target model, which states that a mapping node of type *attr2col* should always be connected to an *FKey* in the target model.

```
fkey(A) :- attr2col(B,A).
```

In contrast to the context analysis of the source model, it is an error now if the ILP engine derives multiple rules for the target model as Assumption 3 would be violated.

In addition to this context analysis, one could also carry out context analysis with reverse roles, which can help constructing reverse transformations, but this step is out of scope for the current chapter.

4.4.3 Learning negative constraints

In a typical model transformation, the existence of a certain mapping structure may depend on the non-existence of certain structures in the source model. ILP systems frequently contain support to identify such negative constraints by means of *constraint learning*. The technique of constraint learning aims at identifying negative constraints of the form *false :- b1, b2, ... , bn.*, i.e. the conjunction of the bodies should never happen.

Learning of negative constraints will be demonstrated on the intuitive mapping rule that only top-level classes should be transformed into database tables. For this purpose, we use an additional prototype mapping model, which is illustrated in Fig. 4.3.

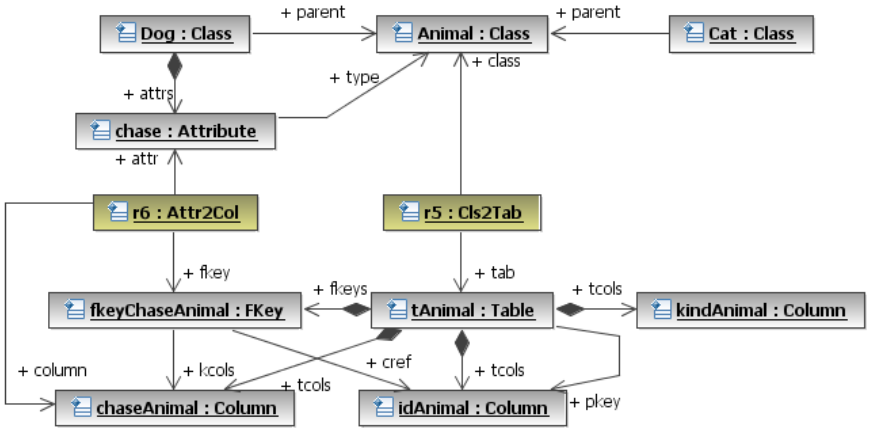


Figure 4.3: Prototype mapping model with unmapped source elements

In case of constraint learning, we only need to construct the background knowledge without positive and negative facts from the source model and the mapping structures.

Example 4.7 Let us continue our running example to handle negative constraints:

```
% Background knowledge
class(animal).
class(dog).
class_parent(dog,animal).
class_attrs(dog,chase).
attribute(chase).
class(cat).
attribute_type(chase,animal).
cls2tab(animal).
```

With appropriate Aleph settings, the following constraints will be induced automatically (which are specific to the mapping node *cls2tab*):

```
false :- cls2tab(A), class_parent(A,A).
false :- cls2tab(A), class_parent(B,A).
```

The first constraint is, in fact, a language restriction of UML (i.e. no class is a parent of itself), while the second derived constraint is a negative constraint of the transformation itself.

A practical problem of the Aleph system we needed to face is that all synthesized constraints are listed instead of listing only the most general ones. Therefore, we need to prune the enumerated list of constraints according to clause entailment in order to keep only the most general constraints.

Example 4.8 For instance, the following constraint is derived by Aleph, but should be filtered out as presenting redundant knowledge with respect to the previous results:

```
false :- cls2tab(A), class_parent(A,A),
         class_parent(B,A).
```

We carry out this pruning by submitting the derived constraints to the Prover9 first-order theorem prover [1]. For this purpose, we present the identified constraints as assumptions to the theorem prover, and the prover tries to construct a formal proof that the more complex constraint implies any of the more simple ones. As for the example above, our assumptions are the following:

$$\forall A \neg(\text{cls2tab}(A) \wedge \text{class\_parent}(A, A)) \tag{4.1}$$
$$\forall A \forall B \neg(\text{cls2tab}(A) \wedge \text{class\_parent}(B, A)) \tag{4.2}$$
$$\forall A \forall B \neg(\text{cls2tab}(A) \wedge \text{class\_parent}(A, A) \wedge \text{class\_parent}(B, A)) \tag{4.3}$$

As a theorem we aim to prove that

$$\forall A \forall B(\text{cls2tab}(A) \wedge \text{class\_parent}(A, A) \wedge \text{class\_parent}(B, A)) \rightarrow$$
$$(\text{cls2tab}(A) \wedge \text{class\_parent}(A, A) \vee \text{cls2tab}(A) \wedge \text{class\_parent}(B, A))$$

Prior to submitting similar problems to the theorem prover, the constraints are ordered according to their length. All (non-filtered) constraints up to length $n-1$ can be used when proving the entailment of a constraint of length n. Fortunately, such theorems are proved immediately by Prover9, therefore, the effort related to the use of sophisticated theorem provers is negligible.

4.4.4 Connectivity analysis

In case of connectivity analysis, we derive different kind of ILP problems for each edge in the target metamodel. The background knowledge B now contains all elements from the source model and all mapping structures as well. This time, the tuple of the mapping structure contains both source and target mappings as follows:

$$ref(src_1, \ldots, src_n, trg_1, \ldots, trg_m).$$

Positive and negative facts (E^+ and E^-) are derived this time from an edge in the target metamodel by deriving a separate ILP problem from each edge type.

Example 4.9 As a demonstration, we carry out the connectivity analysis for target edge *cref*, which is performed (from an ILP perspective) in a similar way as context analysis.

```
% Background knowledge
% Source model
class(employee).
class(clerk).
class_parent(clerk,employee).
class_attrs(clerk,boss).
attribute(boss).
class(manager).
class_parent(manager,employee).
class_attrs(manager,favourite).
attribute_type(boss,manager).
attribute_type(favourite,clerk).
% Mapping model
attr2col(favourite,favouriteClerk,fkeyFavourClerk).
attr2col(boss,clerkBoss,fkeyClerkBoss).
cls2tab(manager,tMngr,idMngr,kindMngr).
cls2tab(clerk,tClerk,idClerk,kindClerk).
% Helper edges
ancestor(X,Y) :- class_parent(X,Y).
ancestor(X,Y) :- class_parent(Z,Y), ancestor(X,Z).

% Positive facts
cref(fkeyClerkBoss,idMngr).
cref(fkeyFavourClerk,idClerk).
% Negative facts = all type consistent pairs
% which are not in positive facts
```

```
cref(fkeyClerkBoss,kindMngr).
cref(fkeyClerkBoss,idClerk).
cref(fkeyClerkBoss,kindClerk).
cref(fkeyClerkBoss,clerkBoss).
cref(fkeyClerkBoss,favouriteClerk).
cref(fkeyFavourClerk,kindMngr).
cref(fkeyFavourClerk,idMngr).
cref(fkeyFavourClerk,kindClerk).
cref(fkeyFavourClerk,clerkBoss).
cref(fkeyFavourClerk,favouriteClerk).
```

Further explanation is needed to understand how negative facts are derived this time. Here, we enumerate all *FKey* and *Column* pairs in the model which are not connected by a *cref* edge. In this step, we impose closed world assumption on our model, thus the set of (type-conforming) object identifiers are exactly those that can be found in the model.

Example 4.10 The Aleph ILP system will derive the following hypothesis for this prototype mapping model.

```
cref(A,B) :- cls2tab(C, D, B, E), attr2col(F, G, A),
             attribute_type(F, C).
```

This rule states that a target node A (of type *FKey*) is connected to a target node B (of type *Column*) with an edge of type *cref*, if class C which belongs to the table D containing B as its primary key is the type of attribute F, which is the source equivalent of the foreign key A.

During a validation phase, domain experts might reveal that this is only a partial solution since *type* edges may lead into a subclass (descendant) of class *E*, and not necessarily into *E* itself. Therefore, if we incrementally refine our knowledge base by adding the prototype mapping model of Fig. 4.3.

Example 4.11 Based upon Fig. 4.3, a new Prolog inference rule is derived in addition to the previous one, which fully corresponds to our expectations:

```
cref(A,B) :-  attr2col(C,A,D), attribute_type(C,E),
              ancestor(E,F), cls2tab(F,G,B,H).
```

This rule extends the previous one by stating that maybe an ancestor F of the class E (which is the type of attribute C) has the corresponding table which stores the mapped primary key column B.

With appropriate additional training for associations, Aleph will derive six rules which "generates" a *cref* edge (using *src* and *dst* instead of *type*, and *asc2tab* instead of *attr2col*).

4.4.5 Generation of model transformation rules

Now we discuss how to derive model transformation rules based upon the inference rules derived by the ILP engine during context analysis and connectivity rules. We use graph transformation rules [51] (Sec. 2.2.2) as the underlying transformation language, which provides a pattern and rule based manipulation technique for graph models frequently used in various model transformation tools. Each rule application transforms a graph by replacing a part of it with another graph. We derive a different set of rules for generating target nodes and edges.

Rules for generating target nodes

The first kind of GT rules that we derive are required for generating all the target nodes. For this purpose, we combine the source and the target context of a certain mapping node as discussed in Alg. 1.

Algorithm 1 An algorithm for generating graph transformation rules to derive target nodes

RN: a node from the mapping metamodel
p_{RN}: a node predicate derived from the mapping metamodel
$SrcCtx$: list of inference rules derived as the source context for RN
$TrgCtx$: list of inference rules derived as the target context for RN
$NegConstr$: negative constraints derived for RN

fun *generate_GT_rule_for_nodes*(RN, $SrcCtx$, $TrgCtx$, $NegConstr$)=
1: **for all** inference rule p^{src}_{RN} *:- a1, a2, an* in $SrcCtx$ **do**
2: create an empty GT rule $R = (LHS, RHS, NAC)$
// Creating the LHS
3: add a node for each variable appearing in the body *a1, a2, an*
4: infer types for the nodes based on the metamodels
5: add an edge to the LHS for each predicate of the body *a1,a2,an*
// Creating the RHS
6: copy LHS to RHS
7: add a mapping node $r1$ for p_{RN}
8: add edges to interconnect $r1$ with all source nodes identified by p^{src}_{RN}
9: **for all** inference rule p^{trg}_{RN} *:- b1* in $TrgCtx$ **do**
10: add a target node n_{b1} to RHS for the body *b1*
11: add an edge to interconnect $r1$ with the new target node n_{b1}
12: **end for**
// Creating NACs based on constraints
13: **for all** negative constraint *false :- n1, nk* in $NegConstr$ **do**
14: add the graph corresponding to the body of the constraint as a new NAC
15: **end for**
// NAC to prevent applying a rule twice on a match
16: add a mapping node $r2$ node to a new NAC
17: add edges to all the source nodes *a1, a2, an* from $r2$
18: **end for**

- The LHS of the rule is constructed for each source context of a mapping node (Lines 2-5).
- The RHS of the rule contains a copy of the LHS and the entire target context of the same mapping node interconnected by an appropriate mapping structure (Lines 6-12).
- A separate NAC is derived for each negative constraint containing a mapping node of a certain type (Lines 13-15).
- Finally, the mapping structure generated by the RHS is added to prevent applying the rule twice on the same source object.

Example 4.12 As a demonstration, we list the graph transformation rule derived from mapping node *cls2tab* in Fig. 4.4. The rule expresses that for each class *C* without a child superclass *CP*, a table *T* is generated with two columns *Id* and *Kind* (which are, in fact, attached to table *T* later on by transformation rules generating edges).

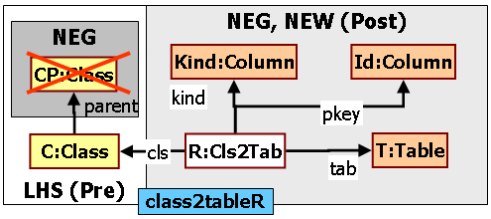


Figure 4.4: GT rule to derive target nodes

Rules for generating target edges

A second set of graph transformation rules aims at interconnecting previously generated target nodes with appropriate edges. For this purpose, we need to combine the Prolog rules derived during connectivity analysis with target contexts. During connectivity analysis, we identify what conditions are required in the source language in order to generate a target edge of a certain type (see Alg. 2).

Algorithm 2 An algorithm for generating graph transformation rules to derive target edges

RE: an edge from the target metamodel
$p_{RE}(A, B)$: an edge predicate derived from the target metamodel
$Connect$: list of inference rules derived connectivity analysis for RE

fun *generate_GT_rule_for_edges*(RE, $SrcCtx$, $TrgCtx$, $NegConstr$)=
1: **for all** inference rule $p^{RE}(A, B)$ *:- a1, a2, an* in $Connect$ **do**
2: create an empty GT rule $R = (LHS, RHS, NAC)$
// Creating the LHS
3: add a node for each variable appearing in the body *a1, a2, an*
4: infer types for the nodes based on the metamodels
5: add an edge to the LHS for each predicate of the body *a1,a2,an*
// Creating the RHS
6: copy LHS to RHS
7: add a target edge $e1$ of type RE leading from node A to node B
// Creating NAC
8: add the same edge as a NAC as well
9: **end for**

- The LHS of such GT rule is constituted from the inference rules derived by connectivity analysis. These inference rules only identify source edges and interconnecting mapping structures, thus the types of the related nodes need to be inferred. (Lines 2-5)
- The RHS is simply a copy of the LHS extended with the corresponding target edge. (Lines 6-7)
- The same edge is added as a NAC is derived to prevent multiple application of the rule on the same match. (Line 8)

Example 4.13 As a demonstration, we present one GT rule (out of the six) derived for generating *cref* edges in Fig. 4.5.

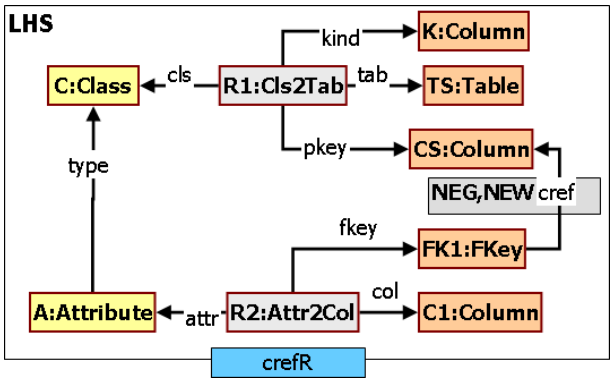


Figure 4.5: A GT rule to derive target *cref* edges

In the current chapter, we systematically separated graph transformation rules into node creation rules and edge creation rules for separation of transformation concerns. It is subject to future research to merge these automatically generated rules in order to obtain a more compact set of transformation rules.

Helper edges can be treated in two ways. A first solution is when each helper edge is added explicitly to the metamodel, thus they are ordinary edges. These edges are then derived by ordinary graph transformation rules. Alternatively, a helper edge can be represented as a recursive graph pattern [J14] (Sec. 3.3.1) which is supported by model transformation frameworks like VIATRA2 or TEFKAT.

Practical assessment of MTBE for the object-relational mapping

As an initial case study for our approach, we implemented the object-relational mapping with MTBE. The source model used in the prototype mapping models contained altogether 10 classes, 3 attributes and 4 associations. Its target equivalent contained 8 tables with 2 or 3 columns for each, and 11 primary key constraints.

Using this relatively small prototype mapping model, we were able to automatically generate 20 graph transformation rules (3 for deriving target nodes and 17 for deriving target edges) by using MTBE, which turned out to be the complete transformation.

Obviously, this prototype mapping model was not created directly from scratch, but it was a result of three iterations. However, all rules have been derived automatically, which exceeded our expectations expressed in [K22].

4.5 KNOWN LIMITATIONS

In order to assess the conceptual and practical limitations of our approach, we have chosen a complex model transformation problem as another case study, where a large abstraction gap needs to be bridged between the source and target languages. The transformation was originally defined in [75], and it aims to derive a Petri net representation of UML statecharts for model analysis purposes.

On the one hand, most of the transformation rules have been generated automatically even for this complex case study. However, two conceptual limitations of the current MTBE approach using ILP techniques have also been revealed. These limitations do not violate our assumptions (see Sec. 4.3.2), but since they arise from practical problems, they demonstrate that these assumptions might be too restrictive for certain model transformation problems.

In order to avoid the presentation of the complex transformation itself, we have taken these problems out of their context and present an analogy in the context of the object-relational mapping.

4.5.1 Non-deterministic transformations

Let us assume that our object-relation transformation requires that each UML class can only be processed if all its parents have already been processed. Thus, we need to define a total order between the classes as an output, which can be denoted by a chain of *next* edges, for instance. Thus the goal is to derive transformation rules for generating this total order using some sample chains as prototype mapping models.

Figure 4.6 depicts one class hierarchy where class *c1* is the parent of class *c2* and *c3*, while class *c4* is not in a *parent* relation with any of the previous classes. One possible ordering of these classes is *c1,c2,c3,c4*, but *c1,c4,c3,c2* is also a possible ordering.

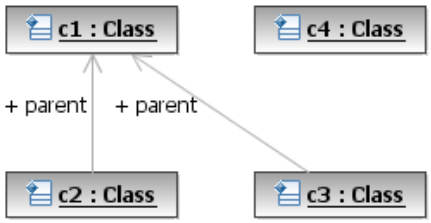


Figure 4.6: A sample class hierarchy for ordering classes

In this case, the transformation has a non-deterministic result, i.e. any total order respecting the partial order imposed by the *parent* relation is a valid result. However, the main problem is that this transformation has two consecutive phases: GT rules generating the first *next* edge are different from GT rules generating subsequent *next* edges. However, these phases were not revealed by the ILP engine.

On the other hand, when the transformation rules are successfully derived, their confluence can be investigated by well-known techniques of graph transformation (such as critical pair analysis [72]).

4.5.2 Counting in transformations

The second problem is related to counting during model transformations. Let us assume that each table in the target database model should have an attribute counting the number of classes it represents. For instance, when a top-level class is transformed into a table, this attribute of the table should count the number of descendants of that class.

When transforming the class hierarchy presented in Fig. 4.6, two tables corresponding to classes *c1* and *c4* are generated. The counter for the corresponding table of *c1* should be set to 3, while the counter related to *c4* is set to 1.

Here the problem is that such a counting is only possible if (i) we use higher-order logic where one can refer to the number of matches for a certain predicate, or (ii) processing the matches sequentially as long as an unprocessed match is found.

4.5.3 Practical limitations

One complexity aspect of the model transformation [75] is that one source node is related to many target nodes due to the abstraction gap between high-level UML models and low-level Petri nets.

As a consequence, both the number of variables (i) and the clause length (*clauselength*) of the hypothesis (which are the most important parameters of the ILP engine concerning performance) had to be kept at a relatively large numbers (above 10). Thus, we experienced cases when the generation of inference rules was not instantaneous (but still completed within 5-10 seconds).

However, more critical performance would be experienced when the source model have the same complexity, i.e. when the derivation of a target edge depends on a large source context with a large number of edges. While our experience shows that this is not so common in a typical model transformation problem, these issues should be kept in mind as potential practical limitations of using an ILP engine for the model transformation by example approach.

It is worth pointing out that Aleph requires at least two positive examples in order to carry out generalization, otherwise only the facts themselves will be retrieved instead of inference rules.

4.6 Prototype Tool Support

We have implemented a prototypical tool chain (illustrated in Fig. 4.7) to automate MTBE by integrating an off-the-shelf model transformation tool with an ILP engine using Eclipse as the underlying tool framework.

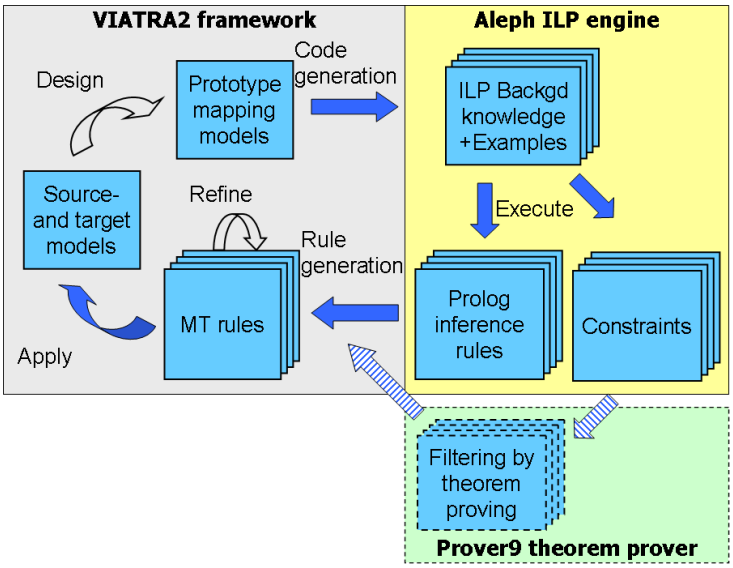


Figure 4.7: A prototype tool chain for automating MTBE

- Source and target metamodels and models as well as prototype mapping models are constructed and stored as ordinary models in the VIATRA2 model space.
- Then a first transformation takes prototype mapping models and generates a set of ILP problems in the way discussed in Sec. 4.4.
- These models are fed into the Aleph ILP engine to induce inference rules (for context analysis or connectivity analysis) or learn negative constraints. Obviously, this step is hidden from the user, as Aleph runs in the background.

- Ongoing work aims at integrating the Prover9 theorem prover in order to filter redundant constraints as described in Sec. 4.4.3.
- Based upon the discovered inference rules, transformation rules are synthesized in the graph transformation based language of the VIATRA2 framework [K1].
- These transformation rules are then executed as ordinary transformations within VIATRA2 to complete the lifecycle of our model transformation by example approach.

This initial tool chain was already a great help for us in carrying out our experiments. Since a different ILP problem is submitted to Aleph for each type of mapping node or target edge, their manual derivation was already infeasible in practice.

However, additional future work should be carried out to improve the usability of the tool chain. Probably the most critical issue is that prototype mapping models need to be defined using the abstract syntax of the language, which is frequently too complex notation for domain experts. Ideally, mappings could be defined using the concrete syntax of source and target languages.

4.7 RELATED WORK

While the current chapter is based on [J1, K22, K23], Strommer et al. independently presented a very similar approach for model transformation by example in [138]. As the main conceptual difference between the two approaches is that [138] presents an object-based approach which finally derives ATL [79] rules for model transformation, while [K22] is graph-based and derives graph transformation rules.

In a recent paper of Strommer [129], their MTBE approach is applied to a model transformation problem in the business process modeling domain and several new MTBE operators used on the concrete syntax were identified. Disregarding the string manipulation (which is obviously out of scope for the current chapter), all the rest can be incorporated in our approach. In this sense, our limitations in Sec. 4.5 only arise in significantly more complex model transformation problems.

Naturally, the model transformation by example approach show correspondence to various "by-example" approaches. Query-by-example [141] aims at proposing a language for querying relational data constructed from sample tables filled with example rows and constraints. A related topic in the field of databases is semantic query optimization [91, 126], which aims at learning a semantically equivalent query which yields a more efficient execution plan that satisfy the integrity constraints and dependencies.

The by-example approach has also been proposed in the XML world to derive XML schema transformers [55,88,110,140], which generate XSLT code to carry out transformations between XML documents. Advanced XSLT tools are also capable of generating XSLT scripts from schema-level (like MapForce from Altova [8]) or document (instance-)level mappings (such as the pioneering XSLerator from IBM Alphaworks, or the more recent StylisStudio [2]).

Programming by example [39, 116], where the programmer (often the end-user) demonstrates actions on example data, and the computer records and possibly generalizes these actions, has also proven quite successful.

While the current chapter heavily uses advanced tools in inductive logic programming [101], other fields of logic programming has also been popular in various model transformation approaches like answer set programming for approximating change propagation in [34] or F-Logic as a transformation language in [64].

The derivation of executable graph transformation rules from declarative triple graph grammar (TGG) rules is investigated in [85]. While TGG rules are quite close to the source and target modeling languages themselves, they are still created manually by the transformation designer.

Inspired by the current results, in the recent years, several groups have started research to come up with new model transformation by example approaches, such as [47, 57, 61, 83, 117, 129, 130].

4.8 CONCLUSIONS

In the current chapter, we proposed to use inductive logic programming tools to automate the model transformation by example approach where model transformation rules are derived from an initial prototypical set of interrelated source and target models. We believe that the use of inductive logic programming is a significant novelty in the field of model transformations.

Let us briefly summarize our experience in using ILP and Aleph. Our experiments carried out on the object relational mapping and a complex model analysis transformation [75] demonstrated that ILP (and Aleph) is a very promising way for implementing MTBE due to the following reasons.

- Partly to our own surprise, we were able to derive all the transformation rules of the object-relational mapping automatically with Aleph, which exceeded our expectations in [K22].
- For rule training, we used relatively small prototype mapping models.
- We used default Aleph settings almost everywhere (only the number of new variables and clauses were set manually). Determination and mode settings were derived systematically when using Aleph for MTBE.

Of course, we experienced certain limitations as well, which were presented in Sec. 4.5. Therefore, our main intentions for future work is to resolve these limitations.

- Non-deterministic transformations were problematic, especially, when an edge of a certain type needs to be generated in a certain order.
- We failed to implement counting in transformation rules, when a target attribute contains the number of matches of a source pattern. This way, handling of attributes values is subject to future work. We expect that data mining techniques could be usable to resolve this issue.

Despite these limitations, the model transformation by example approach has a strong potential, and inductive logic programming turns out to be a powerful tool when implementing it.

Chapter 5

Efficient Execution Strategies for Model Transformations

5.1 Introduction

This chapter focuses on how to provide efficient execution strategies for model transformations. Three high-level strategies will be discussed, namely, model-specific search plans (Sec. 5.2), incremental graph pattern matching (Sec. 5.3) by adapting RETE networks (Sec. 5.4) of expert systems, and dedicated model transformation plugins (Sec. 5.5).

These techniques present several lines of research carried out within the scope of the VIATRA2 project under my scientific leadership. As my contribution primarily lies in the strategies and not in the details, the the current chapter is less detailed compared to other technical chapters of the thesis.

5.2 Model-Specific Search Plans

5.2.1 Introduction

Model transformation tools based on the visual, rule and pattern-based formal paradigm of *graph transformation (GT)* [51, 119] already integrate research results of several decades.

A survey [K28] assessing the performance of graph transformation tools following essentially different approaches on various benchmark examples revealed that approaches (such as Fujaba [58], PROGRES [125] or GReAT [16]) *compiling transformation rules into native executable code* (Java, C, C++) are powerful for model transformation purposes. The performance of the executable code is optimized at compile time by evaluating and optimizing different *search plans* [142] for the traversal of the LHS pattern, which typically *exploits the multiplicity and type restrictions* imposed by the metamodel of the problem domain.

While in many cases, types and multiplicities provide a powerful heuristics to prune the search space, in practical model transformation problems, one has further domain-specific knowledge on the potential structure of instance models of the domain, which is typically not used in these approaches. Furthermore, in case of intensive changes during the evolution of models, the characteristic structure of a model may change as well, therefore a search plan generated a priori at compile time might not be flexible and powerful enough.

In the current section, we introduce **model-sensitive search plan generation** for graph pattern traversal (as an extension to traditional multiplicity and type considerations of existing tools) by *estimating the expected performance of search plans on typical instance models* that

are available at transformation design time. It is worth emphasizing that this technique is directly applicable to furtherly fine-tune the performance of the above mentioned compiler-based GT approaches as well.

Model-specific search plans can be further enhanced with adaptivity [J20] where the optimal search plan can be selected from previously generated search plans at run-time based on statistical data collected from the current instance model under transformation.

5.2.2 Overview of the Approach

The proposed workflow of using these techniques is summarized in Fig. 5.1.

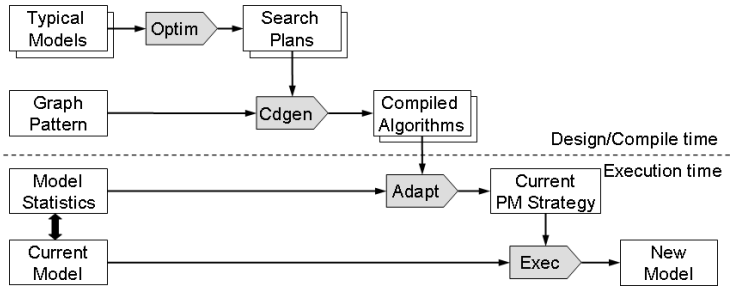


Figure 5.1: Overview of the Approach

Optim First, typical models of the domain are collected (from transformation designers, end users, etc.) from which the optimizer generates one search plan with the best average performance for each typical model.

Cdgen Still at transformation design time, executable (object-oriented) code is generated as the implementation of each search plan.

Adapt At execution time, statistical data is collected on-the-fly from the current model under transformation. Based on this data, a pattern matching strategy (i.e. the implementation of a search plan) can be selected on demand, which yields the best expected performance cost. It is important to ensure that model statistics causes little memory overhead, and the cost of each pre-compiled search plan can also be estimated rapidly.

Exec Finally, the transformation rule is applied on the instance model using the selected pattern matching strategy.

This section focuses exclusively on step *Optim*. Step *Adapt* is detailed in [135, J20], while the detailed discussion of step *Cdgen* is provided in [K2], for instance.

5.2.3 Search plan generation

It is well-known that the most critical step for the performance of graph transformation is the graph pattern matching phase. As pattern matching is determined by only the precondition of a graph transformation rule, we restrict our current investigations only on this part of GT rules.

For this purpose, the generation of *search plans* is a frequently used and efficient strategy. Informally, a search plan defines the order of traversal (a *search sequence*) for the nodes of the instance model to check whether the pattern can be matched.

The search space traversed according to a specific search plan is represented as a **search space tree (SST)** which contains all the decisions that can be made at a certain point during

pattern matching. The root node of a SST represents a partial match as provided by fixing the input parameter nodes of rules. Each path of a SST starting from the root node extends this partial match by matching a fresh (unmatched) node in the pattern.

Below, we present a model-specific search plan generation technique in the following way.

1. First, we introduce the concept of search graphs to obtain an easy to manage representation of GT rule preconditions.

2. Based on the statistics of typical models, model-specific search graphs are prepared by adding numerical weights on the edges of search graphs.

3. The concept of search plans is defined together with a cost function that helps estimate the performance of search plans and formulating when a search plan is optimal.

4. Finally, two algorithms are presented that implement the generation of low cost search plans for model-specific search graphs.

Note that the current section only presents the core ideas of model-specific search plans. This technique has been successfully been extended in [J9, K27] to handle all constructs of the VIATRA2 transformation language (presented in Chapter 3).

Search graphs

In the first phase of the search plan generation process, a search graph is created for each pattern.

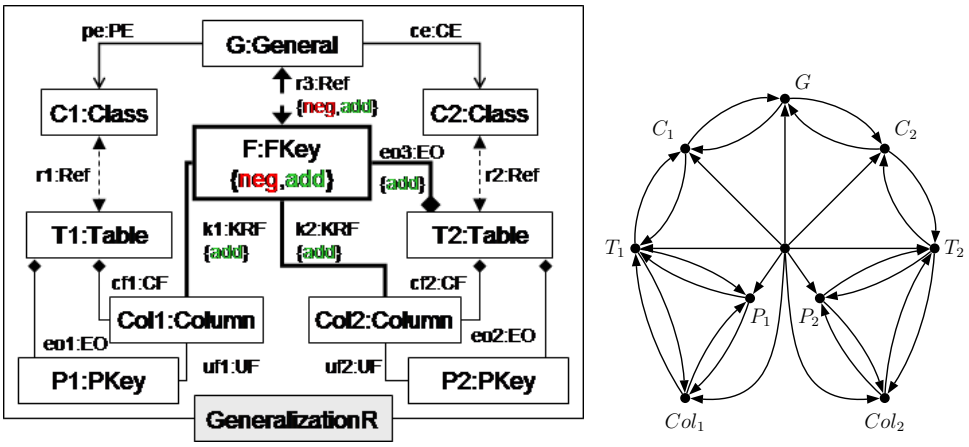
A **search graph** is a directed graph with the following structure. Each node of the graph pattern is mapped to a node in the search graph. We also add a **starting node** to the graph.

1. Directed edges connect the starting node to every other search graph nodes. When such an edge is selected in the search graph for a certain search plan, then the graph pattern matching engine executing this search plan needs to *iterate over all objects in the model of the corresponding type*.

2. Each edge of the pattern is mapped to *a pair of edges* in the search graph that connect the corresponding end nodes in both directions expressing bidirectional navigability.[1] Such an edge can be selected by the pattern matching engine only when the source pattern node is already matched. In this case, the selection of such a search graph edge means a *navigation along the corresponding pattern edge* to reach the unmatched (target) pattern node.

Search graphs for negative application conditions can be handled similarly. In this case, all the matched nodes (i.e. the ones that are shared with LHS graphs) have to be considered as starting nodes. Negative application conditions are typically checked after a complete match has been found for the LHS, but simple checks (e.g. like testing whether edges leaving the shared nodes in the NAC has zero cardinality) can be immediately performed as soon as shared nodes are processed during the traversal of LHS. A more detailed handling of negative application conditions is discussed in [J9].

Example 5.1 A graph transformation rule and its corresponding search graph are depicted in Fig. 5.2(a) and 5.2(b), respectively. The rule itself is taken from an alternate version of the object-relational mapping (Sec. 2.2.3) where each class is transformed into a separate table,



(a) *GeneralizationRule*

(b) Search graph for *GeneralizationRule*

Figure 5.2: A sample graph transformation rule and its corresponding search graph

and generalization (inheritance) is reflected in foreign key constraints as defined in the CWM standard [113].

Nodes and edges of the pattern with *add* annotation have no corresponding elements in the search graph as they denote nodes and edges to be added in the updating phase. Dashed edges in the rule abbreviate the elements of the mapping (traceability) model.

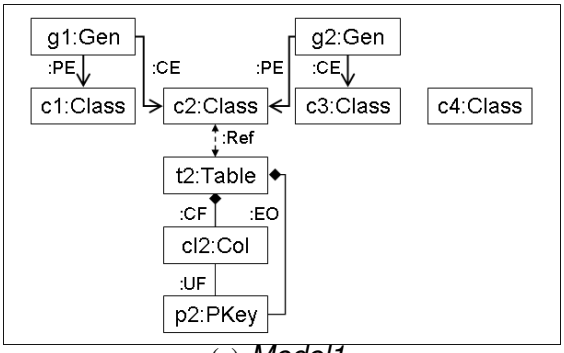
Model-specific search graphs

The initial step for search plan generation takes typical models from the problem domain, e.g. typical UML class diagrams and corresponding database schemas in our case. Node and edge statistics of these typical models are also available, so weights can be defined for the edges of the search graph based on the statistical data collected from a model.

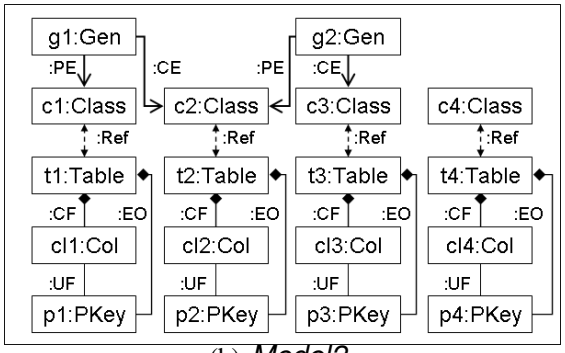
A **weighted search graph** is a search graph with numeric weights on its edges. (Weights are depicted as labels of edges in Figs. 5.3(c) and 5.3(d).) Informally, the weight of an edge can be considered as an average branching factor of a possible SST at the level, when the pattern matching engine selects the given pattern edge for navigation. Such a choice for edge weights provides an easy to calculate cost function that estimates the size of the search space.

Example 5.2 Two models and their corresponding weighted search graphs are depicted in Fig. 5.3. The weight calculation rule is demonstrated on the edge of Fig. 5.3(c) (denoted by a dashed line), which corresponds to the traversal of pattern edge *r2* of type *Ref* in the *Class*-to-*Table* direction. According to our statistics, *Model1* contains 5 *Classes* (since a *Table* is a *Class* in CWM) and 1 reference edge between *Classes* and *Tables*, respectively. As a consequence, if the pattern matching engine matches a *Class* to the pattern node *C2* at some time during the execution, then the probability to find a valid *Table* for pattern node *T2* by navigating along a reference (*Ref*) edge is 0.2 derived by the formula *#Ref(Class,Table)/#Class*. In case of navigation in the opposite direction, the formula can be expressed as *#Ref(Class,Table)/#Table*, thus the corresponding weight is 1.

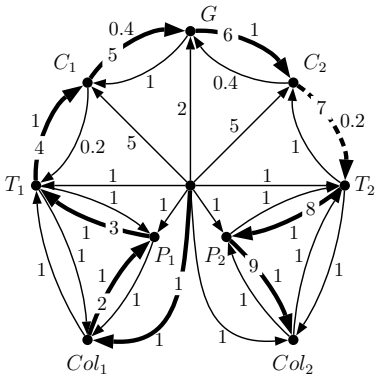
[1] In case of navigability restrictions, only the navigable direction is generated.



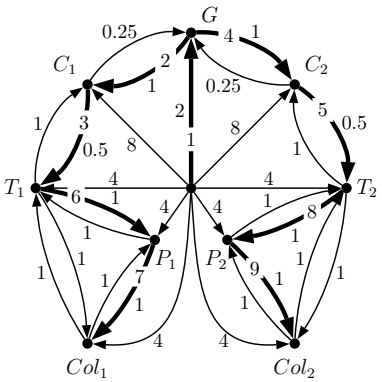
(a) *Model1*



(b) *Model2*



(c) Weighted search graph generated for the *GeneralizationRule* from the statistics of *Model1* and a possible search plan



(d) Weighted search graph generated for the *GeneralizationRule* from the statistics of *Model2* and a possible search plan

Figure 5.3: Sample instance models and corresponding search plans

Search trees and plans

At this point, a weighted search graph is available for each typical model selected by the domain engineer. In this section, first, we introduce the concept of search trees and search plans based on weighted search graphs. Then a cost function is defined for search plans to predict its performance.

A **search tree** is a spanning tree of the weighted search graph. As the starting node has no incoming edges, all other nodes should be reachable on a directed path from the starting node. Edges of a search tree are denoted by thick lines in Figs. 5.3(c) and 5.3(d).The search tree concept can be generalized to handle the completion of partially matched patterns. The generalized concept allows several starting nodes. In this case, a search tree is a forest rooted at starting nodes and they should ensure reachability for all other nodes on edges of trees.

A **search plan** is one possible traversal of a search tree. A traversal defines a sequence in which edges are traversed. The position of a given edge in this sequence is marked by increasing integer numbers written *on* the thick edges in Figs. 5.3(c) and 5.3(d). Two sample search plans (with their corresponding search trees) are shown in Fig. 5.3(c) and 5.3(d).

The **cost of a search plan** (denoted by $w(P)$) is the estimated number of traversed nodes in the corresponding search space tree (SST). The number of nodes at the ith depth-level of the SST is the product of branching factors of edges up to the level i in the search plan, which is $\prod_{j=1}^{i} w_j$, where w_j is the weight of the jth edge according to the order defined by the search plan. The total number of nodes can be calculated by summing the nodes of the SST on a level-by-level basis, which yields to a formula $w(P) = \sum_{i=1}^{n} \prod_{j=1}^{i} w_j$.

Example 5.3 By using this cost function for the search plan of Fig. 5.3(c) on the model of Fig. 5.3(a) and for the search plan of Fig. 5.3(d) on the model of Fig. 5.3(b), we get cost values 5.04 and 8.5, respectively.

As weights denote branching factors, the minimization of a search plan with such a cost function results in a SST that is expected to be optimal in size. Moreover, such a search plan fulfills the first-fail principle criteria as it represents a SST that is narrow at the levels near to its root.

Algorithms for appropriate optimal search plans

Two traditional greedy algorithms are adapted to solve the problems of finding (i) a low cost search tree for a given weighted search graph and (ii) a low cost search plan for a given search tree. Note that traditional algorithms use a different cost function (i.e. the sum of weights) for determining the cost of a spanning tree, which means that their solutions are not necessarily optimal in our case.

For *finding a minimum search tree in a weighted search graph*, the Chu-Liu / Edmonds algorithm [33, 50] is used. This algorithm searches for a spanning tree in a directed graph that has the smallest cost according to a cost function defined as the sum of weights. This algorithm can be outlined in Alg. 3.

Algorithm 3 Finding a minimum search tree in a weighted search graph

Input: Given a weighted search graph with a starting node:

1: Discard the edges entering the starting node.
2: For each other node, select the incoming edge with the smallest weight. Let the selected $n-1$ edges be the set S.
3: If there are no cycles formed by the edges of S, then the selected edges constitute a minimum spanning tree of the graph and the algorithm terminates. Otherwise the algorithm continues.
4: For each cycle formed, contract the nodes in the cycle into a pseudo-node k, and modify the weight of each edge entering node j in the cycle from some node i outside the cycle according to the following equation.

$$c(i,k) = c(i,j) - (c(x(j),j) - min_l\{c(x(l),l)\})$$

5: For each pseudo-node, select the entering edge, which has the smallest modified weight. Replace the edge, which enters the same real node in S by the new selected edge.
6: Go to Step 3 with the contracted graph.

In case of *finding a low cost search plan in a given search tree*, a simple greedy algorithm is used, which is sketched in Alg. 4.

We do not state that these simple algorithms provide *optimal* solutions also for our cost model, but best engineering practice suggests that if edges with weights giving the minimum sum are selected, then the search tree and the search plan consisting of the same edges also have low cost when our special cost function is employed. Simplicity and speed are further arguments in favour of the successful application of such algorithms. Another possibility is to use the only the last element sum [62], which is frequently the dominant factor in the overall cost.

Algorithm 4 Finding a low-cost search plan in a search tree

Ensure: Given a search tree with a starting node.

1: Set the counter to 1 and let S be the set consisting of the starting node.
2: Select the smallest tree edge e that goes out from S.
3: Set the label of e to the value of the counter.
4: Increment the counter by 1 and add the target node of e to S.
5: If the search tree still has a node that is not in S, then go back to Line 2.

5.2.4 Related work

All graph transformation based tools use some clever strategies for pattern matching. Since an intensive research has been focused to graph transformation for a couple of decades, several powerful methods have already been developed.

While many graph transformation approaches (such as [86] in AGG [54], VIATRA [J16]) use algorithms based on *constraint satisfaction*, here we focus on the three most advanced compiled approaches with *local searches using search plans*.

Fujaba [84] performs local search starting from the node selected by the system designer and extending the match step-by-step by neighbouring nodes and edges. Fujaba fixes a single, breadth-first traversal strategy at compile-time (i.e. when the pattern matching code is generated) for each rule. Fujaba uses simple rules of thumb for generating search plans. A sample rule is that navigation along an edge with an at most one multiplicity constraint precedes navigations along edges with arbitrary multiplicity.

PROGRES [142] uses a very sophisticated cost model for defining costs of basic operations (like enumeration of nodes of a type and navigation along edges). These costs are not domain-specific in the sense that they are based on assumptions about a typical problem domain on which the tool is intended to be used. Operation graphs of PROGRES, which are similar to search graphs in the current paper, additionally support the handling of path expressions and attribute conditions. The compiled version of PROGRES generates search plan at compile-time by a greedy algorithm, which is based on the a priori costs of basic operations.

The pattern matching engine of GReAT [137] uses a breadth-first traversal strategy starting from a set of nodes that are initially matched. This initial binding is referred to as pivoted pattern matching in GReAT terms. This tool uses the Strategy design pattern for the purpose of future extensions and not for supporting different pattern matching strategies like in our approach.

Object-oriented database management systems also use efficient algorithms [128] for query optimization, but their strategy significantly differs as queries are formulated as object algebra expressions, which are later transformed to trees of special object manager operations during the query optimization process.

Similar concepts of model-specific search plans have been also investigated and exploited by the GrGen.NET tool [62] independently from our line of research (but one year later in 2006).

5.2.5 Conclusions

In the current section, we proposed a model-sensitive approach for generating search plans for compiled graph transformation approaches. The essence of the technique is to use a priori knowledge obtained from typical designer models. A weighted search graph is derived from statistical data taken from these models. Low cost search plans are defined by tailoring well-known greedy algorithms for the cost function of a traversal.

Model-specific search plans have been implemented in the VIATRA2 and GrGEN.NET [63] graph transformation tools. As VIATRA2 provides a generic model representation (using VPM metamodels), obtaining statistics on model instances turned out to be a complicated practical issues, which had a significant negative effect on the performance of local-search based pattern matching in VIATRA2 . Nevertheless, GrGEN.NET efficiently exploited essentially the same strategy in various tool contests (using a simplified model management framework), which shows the true potential for model-specific search plans.

5.3 INCREMENTAL PATTERN MATCHING

5.3.1 Introduction

The core idea of incremental pattern matching is to make the occurrences of a graph pattern readily available at any time, and they are incrementally updated whenever changes are made. As pattern occurrences are stored, they can be retrieved in constant time (excluding the linear cost induced by the size of the result set itself), which makes pattern matching a very efficient process. Besides memory consumption, the potential drawback is that these stored result sets have to be continuously maintained, imposing an overhead on update operations.

In graph transformation frameworks, pattern matching is required to find the occurrences of left-hand side (LHS) patterns. Since pattern matching can be an important complexity factor in graph transformations, an incremental approach may lead to better performance, especially when transformations are matching-intensive instead of being manipulation-intensive. In this thesis, we provide details for the incremental pattern matcher component of the VIATRA2 framework. It is based on the RETE algorithm, which is a well-known technique in the field of rule-based systems.

5.3.2 Workflow

Initializing an incremental pattern matching engine involves the following conceptual steps:

1. The transformation designer defines various patterns and transformation rules.
2. An incremental pattern matcher (in our case, a RETE network) is constructed based on the pattern definitions.
3. The underlying model is loaded into the incremental pattern matcher as the initial set of matches.

Typically Step 2 and 3 are carried out in incremental pattern matcher as a single, interleaving process. A graph transformation engine with a RETE-based incremental pattern matcher necessitates the repeated execution of the following steps (see Fig. 5.4 for illustration):

1. Match LHS and other patterns in *constant time*;
2. Calculate the difference of the RHS and LHS (and potentially perform more actions);
3. Update the underlying model and notify the incremental pattern matcher of the changes;
4. Propagate the updates within the RETE network to refresh the set of matches.

The initialization of the incremental pattern matcher is not necessarily complete; the pattern matcher RETE network can be freely extended on demand with additional patterns at a later phase. Furthermore, it is worth pointing out that a RETE-based incremental pattern matcher can be integrated with any a graph transformation engine or any other underlying model manipulation library.

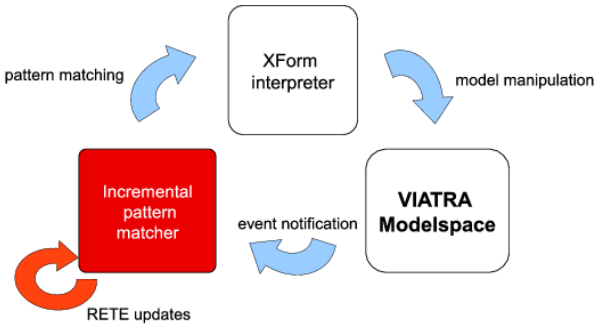


Figure 5.4: Incremental pattern matching information flow

5.3.3 Alternative incremental graph pattern matching approaches

In addition to the RETE-based incremental pattern matcher, we investigated alternative approaches and frameworks for incremental graph pattern matching, which are briefly summarized below.

- **Incremental pattern matching over relational databases.** When models are persisted in a traditional relational database, a straightforward idea is to use database query and manipulation languages to implement graph pattern matching. Relational database management systems offer the concept of triggers to support incrementality, which was investigated [J19] to implement incremental graph pattern matching over relational databases where different database tables and views are notified about changes in other tables.

- **Dedicated incremental pattern matching techniques.** An incremental pattern matching algorithm was proposed in [J21], which introduces specific data structures and notification chains in the presence of both positive and negative patterns, which turned out to be beneficial (compared to local search based techniques) when the patterns became complex.

- **Combination of pattern matching strategies.** In a recent line of our research [J2, K4], we proposed to combine local search (LS) based and incremental graph pattern matching techniques in order to overcome memory limitations of the latter. In this setup, the transformation designers can individually control whether a certain pattern should be matched using incremental or LS-based techniques. Furthermore, when the incremental matching of a pattern runs out of (a dedicated amount of) memory, it can switch to a LS-based mode accelerated by using the partial matches calculated by the incremental matcher.

- **Incremental query evaluation over EMF models.** Our most recent results [K5, K6, K8] aim at porting incremental pattern matching techniques to evaluate complex model queries over large, industry standard EMF models. A new open-source software tool called EMF-IncQuery was also developed for this purpose.

5.4 Incremental Graph Pattern Matching by the RETE Algorithm

The RETE algorithm, introduced in [59], has a wide range of interpretations and implementations. This section describes how we adapted the concepts of RETE networks to implement the rich language features the VIATRA2 graph transformation framework.

In this section, we will gradually construct a RETE-based pattern matcher capable of matching the pattern isTransitionFireable, which is the LHS of the Petri net firing rule depicted in Fig. 5.5 and Listing 5.1.

```
pattern isTransitionFireable(Transition) =
{
  transition(Transition);
  neg pattern notFireable_flattened(Transition) =
  {
   place(Place);
   outArc(OutArc, Place, Transition);
   neg pattern placeToken(Place) =
   {
     token(Token);
     tokens(X, Place, Token);
   }
  }
  or
  {
   place(Place);
   inhibitorArc(OutArc, Place, Transition);
   token(Token);
   tokens(X, Place, Token);
  }
}
```

Listing 5.1: VIATRA source code for the isTransitionFireable pattern

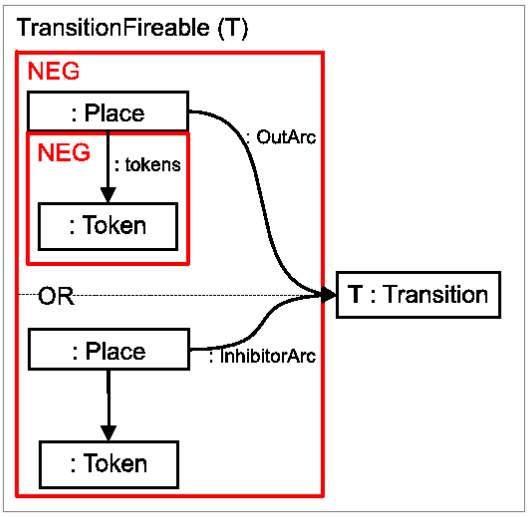


Figure 5.5: Petri-net firing condition

5.4.1 Tuples and Nodes

Partial or complete matches of patterns are stored as tuples in the RETE net. Each node in the RETE net is associated with a (partial) pattern and stores the set of tuples matching the pattern.

The *input nodes* of the RETE serve as the knowledge base representing the underlying model. There is a separate input node for each entity type (class), containing unary tuples representing the instances that conform to the type. Similarly, there is an input node for each relation type, containing ternary tuples with source and target in addition to the identifier of the

edge instance. Miscellaneous input nodes represent containment, generic type information, and other relationship between model elements.

Intermediate nodes store partial matches of patterns. Finally, *production nodes* represent the complete pattern itself. Production nodes also perform supplementary tasks such as filtering those elements of the tuples that do not correspond to symbolic parameters of the pattern (in analogy with the projection operation of relational algebra) in order to provide a more efficient storage of models.

5.4.2 Joining

The join node of a RETE performs a natural join operation on the relations represented by its two parent nodes (linked to the join node by appropriate RETE edges).

Figure 5.6 shows a pattern matcher built for the *sourcePlace* pattern illustrating the use of join nodes. By joining three input nodes, this sample RETE net enforces two entity type constraints and an edge (connectivity) constraint, to find pairs of Places and Transitions connected by an out-arc.

```
pattern sourcePlace(T, P) =
{
  transition(T);
  place(P);
  outArc(A, P, T);
}
```

Listing 5.2: VIATRA source code for the `sourcePlace` pattern

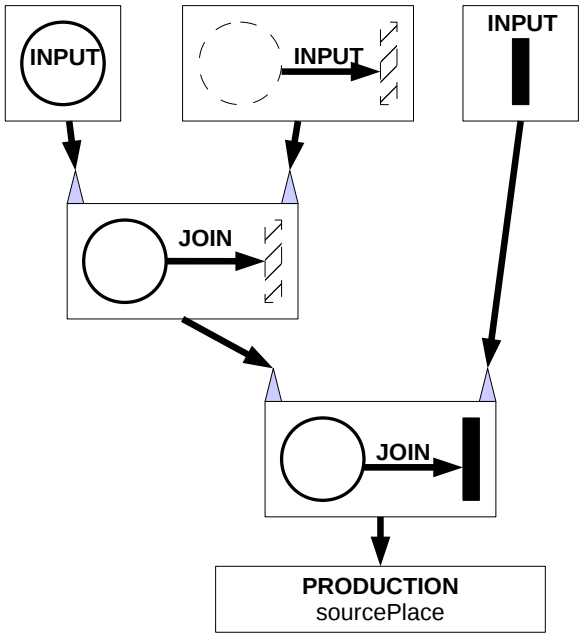


Figure 5.6: RETE matcher for the sourcePlace pattern

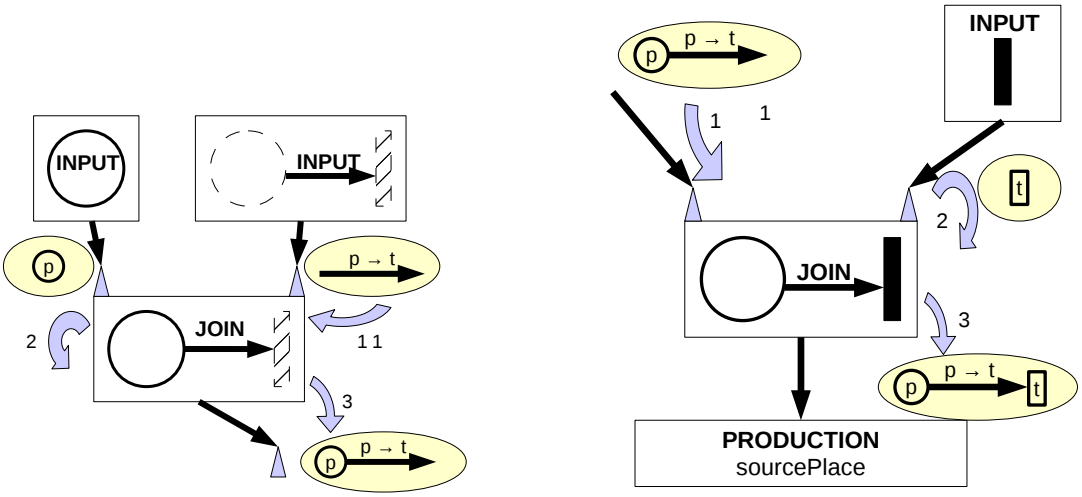
5.4.3 Updates after model changes

The primary goal of the RETE net is to provide incremental pattern matching. To achieve this, input nodes receive notifications about model changes, regardless whether the model was

changed programmatically (i.e. by executing a transformation) or by user interface events.

Whenever a new entity or relation is created or deleted, the input node of the appropriate type will release an update token on each of its outgoing edges. To reflect type hierarchy, input nodes also notify the input nodes corresponding to the supertype(s). Positive update tokens reflect newly added tuples, and negative updates refer to tuples being removed from the set.

Each RETE node is prepared to receive updates on incoming edges, assess the new situation, determine whether and how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes.



(a) Phase I. (b) Phase II.

Figure 5.7: Update propagation

Figure 5.7(a) shows how the network in Fig. 5.6 reacts to a newly inserted out-arc. The input node for the relation type representing the arc releases an update token. The join node receives this token, and uses an index structure to check whether matching tuples (in this case: places) from the other parent node exist. In such a case, a new token is propagated on the outgoing edge for each of them, representing a new instance of the partial pattern “place with outgoing arc”. Fig. 5.7(b) shows the update reaching the second update node, which matches the new tuple against those contained by the other parent (in this case: transitions). If matches are found, they are propagated further to the production node.

5.4.4 Pattern Call

An important feature of the RETE algorithm is that parts of the net can be shared between patterns, thus reducing space and time complexity. The identification of common subpatterns subject to optimization is not yet automated in VIATRA2 , but the transformation designer can assist by decomposing patterns into smaller, reusable parts calling each other (also called pattern composition).

When a pattern calls another pattern, it can simply use the appropriate production node to obtain the set of tuples conforming to the other pattern. Naturally, the production node may have children attached like any other nodes. It is even possible to define recursive patterns that

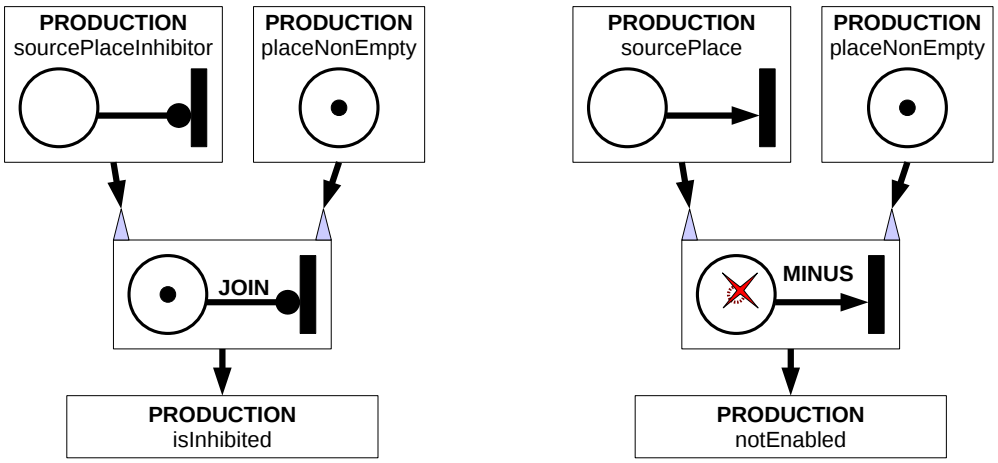
call themselves; in such cases, the production node of the pattern will have an edge leading back to one of the previous nodes. It is the designer's responsibility to ensure that the recursion is well-founded and that there is always exactly one fixpoint as result.

Figure 5.8(a) shows the matcher for pattern isInhibited provided that the simple patterns placeNonEmpty and sourcePlaceInhibitor already have their respective matchers constructed. The matcher selects tuples where the corresponding transition is inhibited by the place for whom the place inhibits the transition, and the place has at least one token.

```
pattern isInhibited(T) = {
  find sourcePlaceInhibitor(T,P);
  find placeNonEmpty(P);
}

pattern notEnabled(T) =
{
  find sourcePlace(T,P);
  neg find placeNonEmpty(P);
}
```

Listing 5.3: VIATRA source code for the `isInhibited` and `notEnabled` patterns



(a) isInhibited (b) notEnabled

Figure 5.8: Positive and negative pattern calls

5.4.5 Negative Application Conditions

The pattern language of VIATRA2 allows to embed patterns into each other as negative application conditions, thus allowing negation at arbitrary depth. To support such negative pattern calls, the existing mechanism for pattern calls can be used, but the production node has to be connected to a negative node instead of a join node. A *negative node* (in the RETE network) has two distinct parents: primary and secondary inputs, respectively. The negative node contains the set of tuples that are also contained by the primary input, but do *not* match any tuple from the secondary input (which corresponds to antijoins in relational databases, see a similar idea with left outer joins e.g. in [J18]).

Figure 5.8(b) shows the matcher for pattern notEnabled, provided that the simple patterns placeNonEmpty and sourcePlace already have their respective matchers constructed. The matcher selects the transitions with source places that do not have any tokens.

5.4.6 Disjunction

OR-Patterns (containing the 'or' keyword) are treated as a disjunction of independent pattern bodies. A separate matcher can be constructed for each body, sharing the production node, which will perform a true union operation on the sets of tuples conforming to each pattern body.

```
pattern isTransitionFireable(T) =
{
  transition(T);
  neg pattern notFireable(T) =
  {
    find notEnabled(T);
  } or
  {
    find isInhibited(T);
  }
}
```

Listing 5.4: Source code for isTransitionFireable pattern

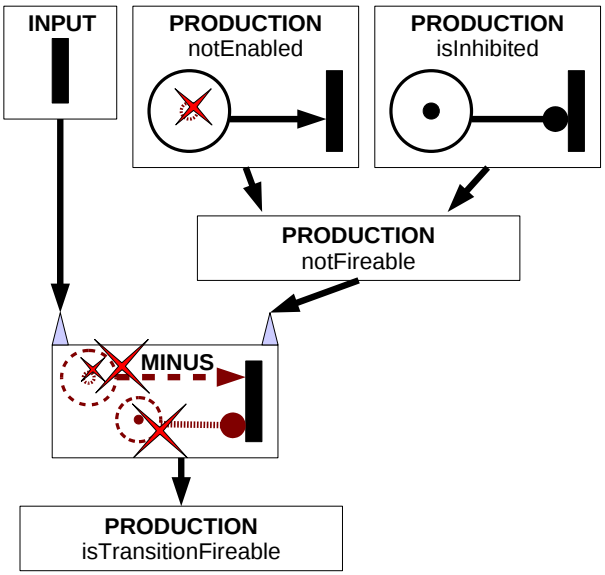


Figure 5.9: RETE matcher for the isTransitionFireable pattern

Figure 5.9 shows the matcher for pattern *isTransitionFireable* (see Listing 5.4), containing an inline negative pattern with two bodies. In this case, each body is a simple reference to a previously constructed pattern, connected to a single production node for the inline pattern.

5.4.7 Term Evaluation

In addition to graph-based structural constraints, the VIATRA2 framework supports the use of attribute conditions to restrict the names and values of model elements. Various arithmetical and logical functions, or even user-provided arbitrary Java code can be applied to model elements to check the validity of a pattern.

The term evaluator node propagates only those tuples that pass a given test. Furthermore, it registers the affected elements of incoming tuples (regardless whether they had passed the filter

or not), so that whenever one of these elements experience change, the tuples containing it can be re-evaluated. If the result changes, the appropriate update tokens will be propagated. The node will monitor changes influencing a tuple until that tuple is finally removed by a negative update received from the parent node.

5.4.8 Experimental evaluation of incremental graph pattern matching

We carried out a series of experiments to evaluate the performance of incremental graph pattern matching in various model transformation scenarios in model driven engineering.

- In our first systematic measurement in [K3], we investigated Petri net simulation and the model synchronization scenario for the ORM problem using incremental graph pattern matching.
- In [J2], the Ant World simulation problem taken from the GRABATS 2008 Transformation Tool Contest was used as a benchmark case study for evaluation. In this setup, we measured the efficiency of combining LS-based and incremental pattern matching strategies.
- In our latest experiments in [K6], we focused on incrementally checking consistency constraints over large AUTOSAR models, which are standard EMF-based model representations of the automotive industry. The actual constraints we evaluated in an incremental way were also taken from the AUTOSAR standard itself.

In Appendix A.2, further details are provided for this last experiment, which is the most recent experimental investigation of ours.

Summary of experiments

All these experiments demonstrated that incremental pattern matching is a very efficient approach even for handling large domain-specific models when (1) model changes are small (e.g. a single transaction does not introduces 1000 model changes), and (2) transactions are not extremely frequent (e.g. transactions do not occur in every 10 ms).

In such a context, the incremental pattern matcher was able to handle queries over models with well over 1 million model elements within fractions of a second as long as the storage of models and the incremental cache fits into memory. In practical problems, memory consumption increased linearly with model size, while incremental query evaluation took constant time. Most importantly, performance is mostly influenced by the size of the change (i.e. the delta between the models), and not the sheer size of the model itself.

In our practical experiments, performance degradation was only experienced when the memory consumption of the incremental cache exceeded the available memory of the executing environment, or when the number of matches for a pattern are highly superlinear compared to the size of the model.

As all of our experiments were carried out on average desktop computers, we strongly expect that with the use of strong server machines, one could very likely handle models well over 10 million elements using the same incremental evaluation technique, thus the first issue can be partially resolved by using a more powerful hardware. For the second issue, we investigated a combined use of local search based and incremental techniques [J2].

Further in-depth analysis using profiler tools revealed that future research efforts should be dedicated to improve model storage and manipulation rather than incremental pattern matching techniques.

5.4.9 Related work

Incremental updating techniques have been widely used in different fields of computer science. Now we give a brief overview on incremental techniques that are used in the context of graph and model transformation.

Incrementality in graph transformation tools

Attribute updates. The PROGRES [123] graph transformation tool supports an incremental technique called attribute updates [74]. At compile-time, an evaluation order of pattern variables is fixed by a dependency graph. At run-time, a bit vector is maintained for each model node expressing whether it can be bound to the nodes of the LHS. When model nodes are deleted, some validity bits are set to false, which might invalidate partial matches immediately. On the other hand, new partial matches are only lazily computed.

View updates. In relational databases, materialized views, which explicitly store their content on the disk, can be updated by incremental techniques like Counting and DRed algorithms [69]. As reported in [J17], these incremental techniques are also applicable for views that have been defined for graph pattern matching by the database queries of [J18]. Later, this approach was extended to incorporate incrementality by triggers on the database level in [J19].

Notification arrays. The paper [J21] proposes a graph pattern matching technique, which constructs and stores a tree for partial matches of a pattern, and incrementally updates it, when the model changes. As a novelty, notification arrays are introduced for speeding up the identification of such partial matches that should be incrementally modified. The main advantage of this solution is that only matches, which appear as leaves of the tree, have to be physically stored, which possibly saves a significant amount of memory. The memory saving technique of [J21] is orthogonal to the structure of the underlying RETE network, and, thus, it can expectedly be used for our approach as well, but the exact integration requires further research and implementation tasks.

RETE networks used for graph transformation.. RETE networks [59], which stem from rule-based expert systems, have already been used as an incremental graph pattern matching technique in several application scenarios including the recognition of structures in images [28], and the co-operative guidance of multiple uninhabited aerial vehicles in assistant systems as suggested by [93]. Our contribution extends this approach by supporting a more expressive and complex pattern language.

Incrementality in constraint evaluation

Incremental OCL evaluation approaches. OCL [107] is a standardized navigation-based query language, applicable over a range of modeling formalisms. Taking advantage of the expressive features and wide-spread adoption of OCL, the project MDT OCL [49] provides a powerful query interface that evaluates OCL expressions over EMF models. However, backwards navigation along references can still have low performance, and there is no support for incrementality.

Cabot et al. [30] present an advanced three step optimization algorithm for incremental runtime validation of OCL constraints that ensures that constraints are reevaluated only if changes may induce their violation and only on elements that caused this violation. The approach uses promising optimizations, however, it works only on boolean constraints, therefore it is less expressive than our technique.

An interesting model validator over UML models [68] incrementally re-evaluates constraint instances (defined in OCL or by an arbitrary validator program) whenever they are affected by changes. During evaluation of the constraint instance, each model access is recorded, triggering a re-evaluation when the recorded parts are changed. This is also an important weakness: the approach is only applicable in environments where read-only access to the model can be easily recorded, unlike EMF. Additionally, the approach is tailored for model validation, and only permits constraints that have a single free variable; therefore general-purpose model querying is not viable.

Incremental Model Transformation approaches

The model transformation tool TefKat includes an incremental transformation engine [71] that also achieves incremental pattern matching over the factbase-like model representation of the system. The algorithm constructs and preserves a Prolog-like resolution tree for patterns, which is incrementally maintained upon model changes and pattern (rule) changes as well.

As a new effort for the EMF-based model transformation framework ATL [80], incremental transformation execution is supported, including a version of incremental pattern matching that incrementally re-evaluates OCL expressions whose dependencies have been affected by the changes. The approach specifically focuses on transformations, and provides no incremental query interface as of now.

VMTS [95, 96] uses an off-line optimization technique to define (partially) overlapping graph patterns that can share result sets (with caching) during transformation execution. Compared to our approach, it focuses on simple caching of matching result with a small overhead rather than complete caching of patterns.

Giese et al. [65] present a triple graph grammar (TGG) based model synchronization approach, which incrementally updates reference (correspondence) nodes of TGG rules, based on notifications triggered by modified model elements. Their approach share similarities with our RETE based algorithm, in terms of notification observing, however, it does not provide support for explicit querying of (triple) graph patterns.

Other rule-based production systems

Improvements and alternatives of the RETE algorithm are now shortly surveyed. In the first two cases, the main goal is to reduce the high memory consumption of the RETE network.

TREAT [99] aims at minimizing memory usage while retaining the incremental property of pattern matching and instant accessibility of conflict sets. Only the input model elements and the (complete) matches are stored, but no memories are used for partial patterns. TREAT is considered faster in certain conditions but less flexible than RETE.

RETE* [139] is a generalization of RETE that attempts to strike a balance between memory size and performance by keeping beta memories stored for frequently used nodes and generating them on-the-fly for the rest. The two extreme cases for the memory retention policy correspond to TREAT and RETE, respectively.

The LEAPS algorithm [20] is claimed to be substantially better than RETE or TREAT at both time and space complexity. The approach can be characterized by lazy evaluation to avoid manifesting tuples unnecessarily, by depth-first firing, and by the introduction of timestamps to set up temporal constraints, which can be used for handling deletion efficiently.

5.5 Model Transformation Plugins

5.5.1 An introduction

As model transformation is becoming a software engineering discipline, conceptual and tool support is necessitated for the entire life-cycle, i.e. the specification, design, execution, validation and maintenance of transformations. However, different phases of transformation design frequently set up conflicting requirements, and it is difficult to find the best compromise. For instance, the main driver in the execution phase is performance, therefore, a *compiled MT approach* (where a transformation is compiled directly into native source code) is advantageous. On the other hand, *interpreted MT approaches* (where transformations are available as models) have a clear advantage during the validation (e.g. by interactive simulation) or the maintenance phase due to their flexibility.

Ideally, advanced model transformation tools should support both interpreted and compiled approaches to separate the *design* (validation, maintenance) of a transformation from its *execution*. Interpreter-based **platform-independent transformations (PIT)** [23, K25] ease the development (and testing, debugging, validation) of model transformations within a single transformation framework without relying on a highly optimized target transformation technology. **Platform-specific transformers (PST)** are compiled (in a complex model transformation and/or code generation step) from an already validated MT specification into various underlying tools or platform-dependent transformation technologies (e.g. Java, XSLT, etc.) and integrated into off-the-shelf CASE tools as stand-alone **model transformation plugins**.

The concepts of platform-independent transformations and platform-specific transformations are first illustrated in the context of In the VIATRA2 framework (see Fig. 5.10).

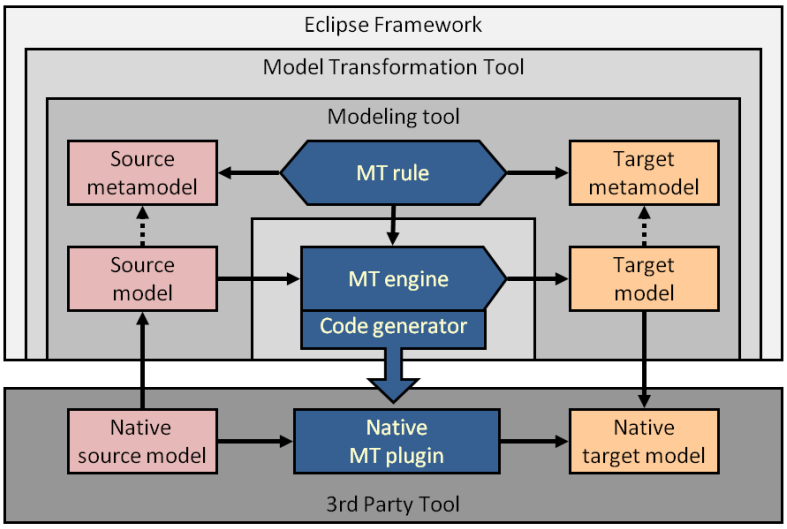


Figure 5.10: Architectural overview of model transformation plugins

Platform-Independent Transformations.. A source user model (which is a structured textual representation such as an XMI description of a UML model exported from a CASE tools) is imported into the VPM model space. Then, platform-independent transformation specifications

can be constructed using model transformation rules (see Chap. 3). The rules are then executed on the source model by the general-purpose VIATRA2 rule interpreter in order to yield the target model. Finally, the target model can be serialized into an appropriate textual representation specific to back-end tools. This way, the transformation is kept within a single transformation framework in order to ease the testing, debugging and validation of model transformations without relying on the highly optimized target transformation technology.

Platform-Specific Transformations.. The VIATRA2 framework also enables the design of meta-transformations that take an already validated transformation specification as the input and yield a platform-specific transformer (e.g., a Java program, or XSLT script) as the output. In other terms, the functionality of a transformation program is *compiled* into a more efficient (but less general) target transformation technology. In this respect, the transformation from a specific source model to its target equivalent can be performed outside the VIATRA2 framework. This is especially important for integrating complex transformations into off-the-shelf CASE tools which, normally, have their own, tool-dependent way for writing transformation add-ons.

The main advantage of this architecture is that the design of model transformations is clearly separated from the execution of model transformations, thus certification of model transformations can be carried out without certifying the complex interpreter itself.

These principles are now instantiated in the business-critical domain. We show how transformation plugins can be implemented for business-critical services built upon the Enterprise Java Beans (EJB3) technology [131]. In addition, the same principles have successfully been applied to generate standalone model transformation plugins for relational databases as underlying host technology in [J17, J18].

5.5.2 Generating model transformation plugins for EJB3

An overview of generating EJB3-specific stand-alone transformer plugins from VIATRA2 transformation descriptions is provided in Fig. 5.11. The general flow of developing and executing model transformations by the VIATRA2 interpreter is the following.

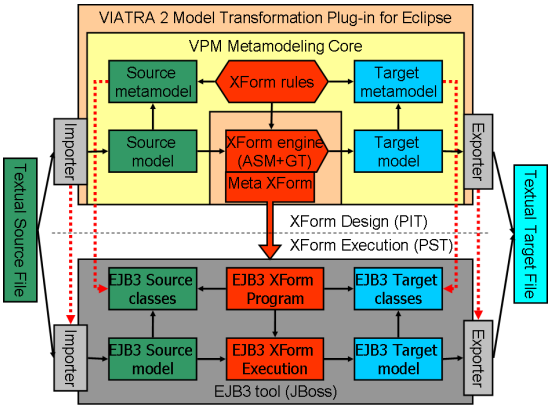


Figure 5.11: Overview of the plugin generation approach

1. **Metamodel design.** Metamodels of the source and target modeling languages (or domains) are designed and stored according to the VPM approach [J15].

2. **Develop importers.** VIATRA2 accepts arbitrary textual source files by offering a flexible way to write model importers.

3. **Model import.** These importers build up an internal VPM representation of the source model which corresponds to its metamodel.

4. **Transformation design.** Model transformations between source and target metamodels are captured by a seamless integration of GT and ASM rules (Chap. 3) [J14].

5. **Transformation execution.** These transformations are executed on the source model by a transformation engine to derive the target model.

6. **Model export.** Finally, the target model can be post-processed by special code generation transformations and code formatters in order to be printed in the required output format.

Standalone model transformation plugins can be generated from VIATRA2 models and transformations for the underlying EJB3 execution platform as follows.

- **Compiling EJB3 models.** *EJB3 entity bean classes* will be generated from the source and target metamodels including the reference objects representing the mapping between them. The persistent storage of EJB3 entity beans will then be handled by the EJB3 application server.

- **Compiling EJB3 transformations.** *EJB3 session beans* will be generated from the MT specification in the form of fully functional EJB business methods. Transaction handling will be provided by the JBoss execution environment to prevent complex transformations from introducing conflicts when manipulating the model in parallel.

- **Reuse of model importers and exporters.** Our experience shows that the development of a model importer also requires significant work. As a consequence, the reuse of VIATRA2 importers (exporters) implemented for building up (extracting) an internal (VPM) model representation from (to) a textual source (target) file in the EJB3-specific transformation plugin is also an important goal. This can be achieved by providing an alternative, EJB3-specific implementation of the VIATRA2 model manipulation which redirects and translates VIATRA2 specific calls to EJB3-specific calls.

A detailed description of the derivation process is out of scope for the current thesis, the interested reader is referred to [K2]. Moreover, the graph pattern matching part in EJB3 plugins were further optimized in [136].

5.5.3 Related Work

While there is already a large set of model transformation tools available in the literature using graph rewriting, below we focus on providing a brief comparison with the most popular *compiled approaches* that show conceptual similarities with our work. A more detailed comparison on the difference in the applied graph transformation strategies can be found in [J2, J20, K4].

Fujaba [103] compiles visual specifications of transformations [58] into executable Java code based on an optimization technique using search graphs with a breadth-first traversal strategy penalizing many-to-many multiplicity constraints. Our approach is different from Fujaba in the use of EJB3 beans instead of pure Java classes and the model-sensitive generation of search plans (see [J20] for details).

PROGRES [125] supports both interpreted and compiled execution (generating C code) of programmed graph transformation systems. It uses a sophisticated cost model for defining a priori costs of basic operations (like the enumeration of nodes of a type and navigation along

edges) for generating search plans. Our solution was, in fact, influenced by PROGRES; however, our model-specific cost function provides a dynamic (transformation-time) estimation for the complexity of matching graph transformation rules.

The pattern matching engine of compiled GReAT [137] (generating C++ code) uses a breadth-first traversal strategy starting from a set of nodes that are initially matched. Such a C++ solution typically provides an efficient solution for compiled transformations when integrated into embedded systems. In contrast, our transformation plugins primarily target web-based target platforms and achieve high performance as being deployed on application servers.

OPTIMIX [11] is a tool for generating algorithms in C or Java which construct and transform directed relational graphs with a special focus on tasks in program compilation and optimization. OPTIMIX supports edge addition rewrite systems (EARS) and exhaustive graph rewrite systems (XGRS) using an input language equivalent to a subset of Datalog.

The idea of separating transformation design from transformation execution also appears in the MOLA environment [81] by providing an Eclipse EMF-based execution environment. While EMF-based models and EJB3 models show conceptual similarities concerning their structure, EJB3 provides additional support for important dynamic features such as e.g. transaction handling.

5.6 Conclusions

The main goal of the current chapter was to provide efficient execution strategies for model transformation techniques.

Model-specific search plans exploited the fact that the traversal order of graph patterns can be tailored to the structure of the underlying model using model-level statistical information. In our experiments, we found that this technique is most efficient when combined with adaptivity, i.e. when the actual search plan is selected at execution time. Obviously, this techniques relies upon the cheap retrieval of statistical information on the actual model, which may or may not be provided by state-of-the-art model management and DSML frameworks.

Incremental graph pattern matching aimed at caching the matches of graph patterns, and then maintaining these caches as the underlying model changes. Incremental graph pattern matching accelerates model transformations by orders of magnitude when the match set is relatively small compared to the size of the underlying model and the size of the change. This is typical in many model simulation and model synchronization scenarios of MDD. Performance issues can be experienced mainly by increased memory consumption when the match set of a pattern is large, e.g. when calculating transitive closure, for instance.

Model transformation plugins allow to embed the model transformations into industrial context by separating transformation design from transformation execution. As a consequence, model transformations can be executed in third party tools (without the presence of the transformation development framework, like VIATRA2), which is a significant practical advantage. However, developing a new model transformation plugin is still a high effort task, thus future research should investigate how this effort can be reduced by increasing reusability on different levels.

dc_244_11

Chapter 6

Termination Analysis of Model Transformations

6.1 Introduction

Many researchers and practitioners have recently revealed that model driven software development relies not only on the precise definition of modeling languages taken from different domains, but also on the unambiguous specification of transformations between these languages. Graph transformation (GT) [38, 119] has been applied successfully to many model transformation (MT) problems. Many success stories were in the field of model analysis which aim at projecting high-level UML models into mathematical domains by model transformations to carry out formal analysis.

A core problem of model driven engineering is related to the *correctness of model transformations*, namely, to guarantee that certain semantic properties hold for a *trusted model transformations*. For instance, when transforming UML models into mathematical domains, the results of a formal analysis can be invalidated by erroneous model transformations as the systems engineers cannot distinguish whether an error is in the design or in the transformation. It is possible that transformation rules interfere with each other and thus they may cause semantic problems, which is not acceptable for trusted model transformations.

Most typical correctness properties of a trusted model transformation are termination, uniqueness (confluence) and behaviour preservation. In the current chapter, we provide a Petri Net based technique for the *termination analysis of model transformations specified by GTSs*. As termination is undecidable for graph grammars in general [112], we propose a sufficient criterion, which either proves that a GTS is terminating, or it yields a "maybe nonterminating" (do not know) answer.

The essence of our technique is to derive a simple Petri net which simulates the original GTS by abstracting from the structure of instance models (graphs) and only counting the number of elements of a certain type. If we manage to prove by algebraic techniques that the Petri net runs out of tokens in finitely many steps regardless of the initial marking, then we can conclude that the original GTS is terminating due to simulation. In order to handle graph transformation systems with negative application conditions as well, we introduce the notions of forbidden and permission patterns, and overapproximate how different rules influence each other when generating permissions.

As the derived Petri net model is of managable size (comparable to the number of elements in the metamodels), our technique can yield positive results for judging the termination of vari-

ous model transformation problems captured by graph transformation techniques.

The rest of the chapter is organized as follows. Based upon the preliminaries of graph transformation systems (Sec. 2.2.2) and Petri nets (Sec. 2.4), Sec. 6.2 proposes a P/T net abstraction of GTS with rules having negative application conditions (NAC). Sec. 6.3 presents sufficient conditions for termination of GTSs by solving algebraic inequalities. Our approach will be exemplified using the GTS of the object-relational mapping (see Sec. 2.2.3). Finally, Sec. 6.4 discusses related work and Sec. 6.5 presents our conclusions and proposals for future work.

6.2 A Petri Net Abstraction of Graph Transformation

6.2.1 Definition of the core abstraction

First we map a graph transformation system *without negative application* conditions into a Petri net (which is called *cardinality (P/T) net* in the sequel) by only keeping track of the number of objects in the model graph (separately for each node and edge in the type graph) but abstracting from the structure of the model graph.

Informally speaking, since the LHS of a GT rule requires the presence of nodes and edges of certain types, the derived transition removes tokens from all the places storing the instances of the corresponding types. Furthermore, the RHS of a GT rule guarantees the presence of nodes and edges of certain types, thus the derived transition generates tokens for the places storing the instances of such types. Later we show that this is a proper abstraction, i.e. the derived P/T net simulates the original GTS, i.e. when a GT rule is applicable, the corresponding transition in the P/T net can be fired as well.

Definition 6.1 (Mapping from GTS to P/T nets) The mapping $\mathcal{F}(GTS) = (\mathcal{F}_{TG}, \mathcal{F}_G, \mathcal{F}_R) \rightarrow PN$ (where $GTS = (R, TG)$ and $PN = (P, T, E, w)$ with initial marking M_0) is formally defined as follows:

- $\mathcal{F}_{TG} : TG \rightarrow P$*: Types into places.* For each node and edge $y \in N_{TG} \cup E_{TG}$ in the type graph TG, a corresponding place $p_y = \mathcal{F}(y)$ is defined in the cardinality P/T net.
- $\mathcal{F}_G : G \rightarrow M_0$*: Instances into tokens.* For each node and edge $x \in N_G \cup E_G$ in an instance graph G with type $y = type(x)$, a token is generated in the corresponding marking $M_G = \mathcal{F}(G)$ of the target P/T net. Formally, for all places $p_y = \mathcal{F}(y)$, the marking of the net is defined as $M_G(p_y) = card(G, y)$.
- $\mathcal{F}_R : R \rightarrow (T, E, w)$*: Rules into transitions.* For each rule r in the graph transformation system GTS, a transition $t_r = \mathcal{F}(r)$ is generated in the cardinality P/T net such that
  - *Left-hand side:* If there is a graph object x in L with $y = type(x)$, then an incoming arc (p_y, t_r) is generated in the P/T net where $p_y = \mathcal{F}(y)$ and the weight of the arc $w(p_y, t_r)$ is equal to the number of graph objects in L of the same type y. Formally, if $\forall x, y : x \in L \wedge y = type(x) \wedge \mathcal{F}(y) = p_y \Rightarrow (p_y, t_r) \in E \wedge w(p_y, t_r) = card(L, y)$.
  - *Right-hand side:* If there is a graph object x in R with $y = type(x)$, then an outgoing arc (t_r, p) is generated in the P/T net where $p_y = \mathcal{F}(y)$ and the weight of the arc $w(t_r, p_y)$ is equal to the number of graph objects in R of the same type y. Formally, if $\forall x, y : x \in R \wedge y = type(x) \wedge \mathcal{F}(y) = p_y \Rightarrow (t_r, p_y) \in E \wedge w(t_r, p_y) = card(R, y)$.

Example 6.2 In Fig. 6.1 rule *liftAssocDstR* of the example in Fig. 2.6 is shown on the left together with the corresponding transition *liftAssocDstR* (on the right) of the P/T net abstraction of the example. Note that indices of $\mathcal{F}()$ will be omitted for simplicity.

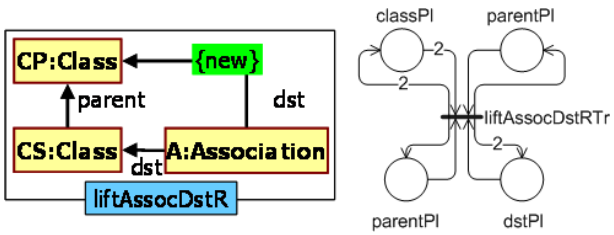


Figure 6.1: Transition corresponding to rule liftAssocDstR

As the GT rule *liftAssocDstR* contains two *Class* nodes, one *Association* node, one *parent* edge and one *dst* edge, the corresponding transition is enabled if the corresponding type places (with identical labels) contain at least 2, 1, 1, and 1 tokens, respectively. Since the application of the rule preserves all items and creates one *dst* edge, the firing of transition *liftAssocDstR* puts 2, 1, 1, and 2 tokens to these places, respectively.

Note, however, that the transition of Fig. 6.1 is always enabled and thus, it would directly cause non-termination. Therefore, we now extend our abstraction technique to handle graph transformation rules with negative application conditions as well, which are frequently used in model transformation problems.

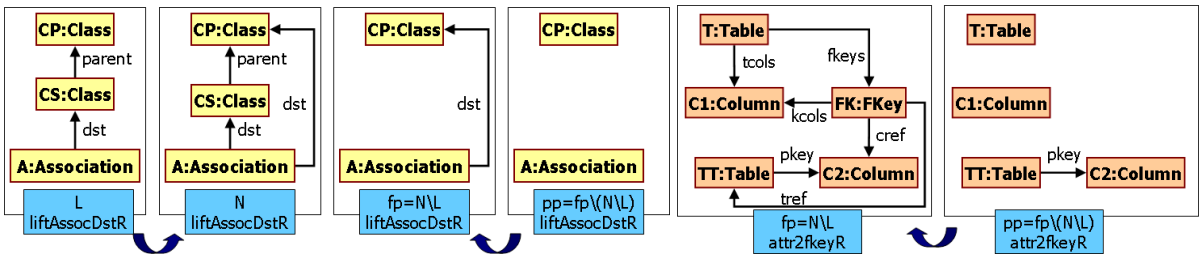
6.2.2 Extensions for negative conditions

Permission places. In order to cope with NACs, the P/T net is extended with *permission places* to restrict the firing of a transition. We add one permission place for each NAC in the GTS, and the idea of a permission place is to count how many times the GT rule can be applied to the current instance graph (such that the corresponding NAC does not violate these matches).

- *Start graph.* The initial marking of permission places shall enable the firing of a transition as many times as the corresponding GT rule is applicable to the start graph by giving a permission token.
- *Removing permissions.* If a new match of some NAC N_i of a GT rule r is generated or an existing match of the LHS of the same rule r is destroyed by the application of some GT rule r' then one or more tokens should be removed from the permission place corresponding to N_r^i.
- *Creating permissions.* If an existing match of the NAC of a GT rule r is destroyed or a new match of the LHS of the same rule r is generated by the application of some GT rule r' then one or more tokens should be generated to the permission place related to N_r^i.

Unfortunately, the exact number of tokens created for or removed from a permission place depends on the actual graph structure. Therefore, we cannot derive a constant weight *a priori* for the corresponding arcs in the P/T net; instead we write $w(G)$ on such arcs to denote that the weight of the arc is dependent on graph G. However, we know that such an arc weight $w(G)$ is finite, i.e. we can only generate and remove a finite number of new permissions for any permission place.

Overapproximation for permissions. Therefore, we need to define an overapproximation of the potential number of rule applications, which still simulates the GTS, yet it is precise enough to detect termination for a certain class of model transformation problems.



(a) Rule *liftAssocDstR* (b) Rule *attr2fkeyR*

Figure 6.2: Forbidden and permission patterns

- In our proposal, we only remove one token from a permission place when it is absolutely guaranteed (by analyzing the original GT rule) that a *permission should be destroyed* each time the rule is applied. In case of GT rules with NAC, such a situation is when a GT rule cannot be applied on the same match twice due to a NAC.

- In case of *generating a permission*, we should consider all possible values for the arc weight $w_i(G)$, thus we create a new variable c_i which runs over positive integers.

Permission and forbidden patterns. An initial idea for granting permissions is to consider the causalities of GT rules, i.e. when a rule generates a new match for another rule, a new permission is generated as well. However, this solution is unable to handle cases when GT rules are generating a bounded number of new matches for themselves (i.e., when a rule is causally dependent on itself).

For instance, each application of rule *liftAssocDstR* (in Fig. 6.1) generates a new *dst*, thus a new match for itself, which seems to be a direct cause for non-termination. On the other hand, if the meaning of a permission is related to the number of *Class-Association* pairs not connected by a *dst* edge, we notice that this number is strictly decreasing, thus no new permission is granted by GT rule *liftAssocDstR* for itself. This insight is captured formally by *forbidden and permission patterns*.

Definition 6.3 (Forbidden and permission pattern) Let $GTS = (R, TG)$ be a graph transformation system. A **forbidden pattern** fp_r^i is defined for each NAC N_r^i of rule r as the smallest subgraph of N_r^i that contains $N_r^i \setminus L_r$ (also called as the *context* of $n^i : L_r \to N_r^i$).

The **permission pattern** pp_r^i (of the same NAC N_r^i) is defined as smallest subgraph of fp_r^i that contains $N_r^i \setminus L_r$ (also called as the *boundary* of $n_r^i : L_r \to N_r^i$), which is defined formally as $fp_r^i \setminus (N_r^i \setminus L_r)$.

Informally, the permission pattern can be interpreted as an LHS pattern having a NAC with the forbidden pattern. The exact number of permissions for a rule is calculated as the number of matches of the permission pattern having the forbidden pattern as a NAC.

Example 6.4 The concepts of forbidden and permission patterns are demonstrated in Fig. 6.2(a). The forbidden pattern (*FP*) of rule *liftAssocDstR* contains a *dst* edge leading from Association *A* to Class *CP*. Here $N \setminus L$ contains the single *dst* edge while the two nodes are added to guarantee that the forbidden pattern forms a graph. In order to obtain the permission pattern (*PP*), we simply remove this *dst* edge from the forbidden pattern.

Definition of cardinality P/T with permission places. We now formally define cardinality P/T nets with permission places.

Definition 6.5 (Cardinality P/T net with permission places) The **cardinality P/T net with permission places** of GTS is a $PN = (P, T, E, w)$ derived by the mapping $\mathcal{F}_{pp}(GTS)$ by extending $\mathcal{F}(GTS)$ in the following way:

- *Variables as weight functions.* We extend the weight function of a P/T net to $w : E \to \mathbb{N}^+ \cup V$ where V is a set of variables ranging over $\mathbb{N}^+$.
- *NACs into permission places.* For each NAC N^i of a rule r a corresponding permission place $p_{r_{N^i}} = \mathcal{F}_{pp}(r_{N^i})$ is defined in the cardinality net.
- *Matches of permission patterns into tokens (initial marking).* For each NAC N^i of a rule r as many tokens are generated in the corresponding permission place as the number of injective matches m of permission pattern pp_r^i in the instance graph G which satisfies the derived NAC $pp_r^i \to fp_r^i$, (i.e., there is no injective match of the forbidden pattern fp_r^i to G along m).
- *NACs into pre arcs.* For each rule r with NACs $N^1, \ldots, N^k$, if there is an injective morphism $k_i : N^i \to R$ compatible with r for some NAC N^i (informally, everything included in the NAC N^i exists or it is created by the RHS), an incoming arc $(p_{r_{N^i}}, t_r)$ is generated in the P/T net with weight 1.
- *Rule actions into post arcs.* For each pair of rules $r = (L_r \xleftarrow{l} K_r \xrightarrow{r} R_r)$ with NACs $N^1, \ldots, N^k$ and $r' = (L'_r \xleftarrow{l} K'_r \xrightarrow{r} R'_r)$, an outgoing arc $(t_{r'}, p_{r_{N^i}})$ $(i : 1 \leq i \leq k)$ is generated in the P/T net (i.e. from the transition of rule r' to the permission place of r_{N^i}) with a *variable* arc weight $v_{r',r_{N^i}}$ if
  1. at least one graph object o is deleted by r' (from the forbidden pattern fp_r^i of r) such that there exists a graph object $o' \in N^i \setminus L_r$, and $type(o) = type(o')$ or
  2. at least one graph object o is created by r' such that there exists a graph object $o' \in pp_r^i$, and $type(o) = type(o')$.

Informally, instead of regarding the causality between two rules based upon the RHS of rule r' and the LHS of r, we define causality between the effects of a rule r' and the permission pattern of r.

Furthermore, in order to overapproximate the graph dependent arc weights $w(G)$, we introduce variables as weights for such arcs. As a consequence, for each step of the P/T net, we can substitute the variables with proper values to simulate the original GTS in a step-wise way. In order to prove termination later in Sec. 6.3, we will show that any substitution of these variables fulfill certain algebraic properties.

The **incidence matrix** of the P/T net abstraction of GTS with NACs is denoted as $W(\underline{v})$, which notation emphasizes that W contains variables at locations where new permissions are generated for a rule.

Example 6.6 The incidence matrix of the GTS for the object-relational mapping (see Sec. 2.2.3) is given in Fig. 6.3. The places (columns) refer to the type places corresponding to the type graphs of Fig. 2.2 and Fig. 2.3, while transitions (rows) refer to corresponding rules of Fig. 2.6. The right-most columns of the matrix denote permission places.

	UML								Ref								
	Assoc	Class	Attr	src	dst	attrs	type	par	t2a	t2c	c2a	A2T	C2T	A2C	a2t	c2t	a2c
parentClosureR	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
liftAttrsR	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
liftAttrTypeR	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
liftAssocSrcR	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
lisftAssocDstR	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
class2tableR	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0
assoc2tableR	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0
attr2columnR	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1
attr2fkeyR	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
assoc2fkeyR	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	DB									permission places (for each NAC)										
	Tab	FKey	Col	tref	fkeys	pkey	tcols	cref	kcols	parCl	lAttr	lAttrT	lAssrc	lAsDst	cl2ta1	cl2ta2	as2tab	attr2c	at2fk	as2fk
parentClosureR	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0
liftAttrsR	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0
liftAttrTypeR	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0
liftAssocSrcR	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0
lisftAssocDstR	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0
class2tableR	1	0	2	0	0	1	2	0	0	0	0	0	0	0	0	-1	0	0	v_1	v_2
assoc2tableR	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	v_3	v_4
attr2columnR	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	v_5	v_6
attr2fkeyR	0	1	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	-1	0
assoc2fkeyR	0	2	2	1	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	-1

Figure 6.3: Incidence matrix of the P/T net abstraction

Note that the incidence matrix is independent of the initial marking of the cardinality P/T net, thus our termination result is valid for any initial marking. It is worth pointing out that the proposed abstraction highly relies on the fact that a RHS contains at least one of its NACs. Note that this is typical for model transformation problems where NACs are frequently used to prevent the application of a rule multiple times on the same match.

Example 6.7 (Cardinality P/T net with permission patterns) Rule *parentClosureR* (see Fig. 2.6) generates new *parent* edges, which are required for the match of rule *liftAssocDstR* (see Fig. 2.6), thus the two rules are causally dependent. However, no new permissions are generated for the latter, since rule *parentClosureR* should remove a *dst* edge (see the forbidden pattern) or create new *Class* or *Association* nodes (see the permission pattern) for a new permission to be generated (see the permission and forbidden patterns in Fig. 6.2(a)).

On the other hand, rule *class2tableR* generates new permissions for rule *attr2fkeyR*, since the tables created by the former are present in the permission pattern of the latter (which consists of tables *T*, *TT* and columns *C1* and *C2*, see Fig. 6.2(b)). Consequently, a variable v_1 is used as the weight of the corresponding arc leading from the transition of *class2table* to the permission place of *attr2fkeyR*.

6.3 Termination Analysis of Graph Transformation

Now we propose a termination analysis for GTS using a generalization of non-repetitiveness results from P/T nets [102].

A P/T net is *partially repetitive* if there exists a marking M_0 and a firing sequence s from M_0 such that some transition occurs infinitely many times in s. Furthermore, a main result from P/T net theory states that a P/T net with the incidence matrix W is partially repetitive if and only if there exists a Parikh–vector $\underline{\sigma} \geq \underline{0}, \underline{\sigma} \neq \underline{0}$ such that $W^T \cdot \underline{\sigma} \geq \underline{0}$. As a consequence, if a P/T net is not partially repetitive (i.e., no Parikh–vector $\underline{\sigma} \geq \underline{0}, \underline{\sigma} \neq \underline{0}$ exists that satisfies $W^T \cdot \underline{\sigma} \geq \underline{0}$), then only finite firing sequences exist from any intial marking M_0, which proves termination.

Our generalization lies in the fact we do not require the existence of the incidence matrix W. Instead we state that if sequences of state vectors fulfill the condition that at least one component of the state vector is decreasing (wrt. each previous state vector in the sequence) in each step it guarantees that the $\underline{0}$ state is reached in finite steps. Our reason for this generalization is that W may contain variables at permission places.

Lemma 6.8 If for all infinite sequences $\{M_i\} = M_0, M_1, \ldots$ of n-dimensional (state) vectors of nonnegative integer values with $M_j - M_{j-1} < \infty$ for all j

$$(1) \quad \forall i, \forall j : j > i, M_i \not\equiv \underline{0} \Rightarrow \exists k : M_j[k] - M_i[k] < 0, \text{and}$$

$$(2) \quad \forall i, \forall j : j > i, M_i \equiv \underline{0} \Rightarrow M_j \equiv \underline{0}$$

then $M \equiv \underline{0}$ in finitely many steps, i.e. $\exists s : M_s \equiv \underline{0}$ (where $M_j[k]$ denote component k in vector M_j).

PROOF In the following we use the following indices: $i, j \in \mathbb{N}$ to refer to the elements of the sequence, and $k, l \in \{1..n\}$ to refer to the components of a vector in the sequence.

- First, observe, that if the properties (1) and (2) hold for an infinite sequence then the properties hold also for any subsequence of this sequence.
- In this case, we state that there should exist an infinite subsequence $M_{y_0}, M_{y_1}, \ldots$ (of $M_0, M_1, \ldots$) with $M_{y_0}[k] = M_{y_1}[k] = M_{y_2}[k] = \ldots$ for some component $k \in \{1..n\}$, i.e. the subsequence at component k is constant.

  To prove this, let us construct this subsequence.

Step 1. Let us define the following nonnegative difference–vector for all i, j:

$$M_{[i,j]}[k] = \begin{cases} M_i[k] - M_j[k], & \text{if } M_i[k] - M_j[k] \geq 0 \\ 0, & \text{else} \end{cases}$$

Step 2. Then let us examine the infinite sequence $\{M_{[0,i]}\}$ of difference–vectors, i.e. the above defined difference of all vectors from M_0.

Step 3. Since all components of M_0 are finite, there exists an upper bound $K = max\{M_0[k]\}$ that is also an upper bound for the components of the difference vectors according to the definition. Thus there may exist at most K^n different difference-vectors. Since we have an infinite sequence of difference-vectors, there exists (by the Pigeonhole Principle) an infinite subsequence of difference-vectors $M_{[0,y_0]}, M_{[0,y_1]}, \ldots$ such that $M_{[0,y_0]} = M_{[0,y_i]}$ for all i, where y_0 is a finite index.

Step 4. According to property (1) of $\{M_i\}$, difference-vector $M_{[0,y_0]}$ has at least one component l_1 which is positive, i.e., $M_{[0,y_0]}[l_1] > 0$ (otherwise $M_i \equiv \underline{0}$ is reached).

From Step 3, we know that $M_{[0,y_0]}[l_1] = M_{[0,y_i]}[l_1]$ for all i. Since $M_{[0,y_0]}[l_1] > 0$ and $M_0[l_1] - M_{y_0}[l_1] = M_{[0,y_0]}[l_1] = M_{[0,y_i]}[l_1] = M_0[l_1] - M_{y_i}[l_1]$, therefore $M_{y_0}[l_1] = M_{y_i}[l_1]$ for all i. Thus the infinite subsequence $\{M_{y_i}\}$ is constant at component l_1.

- Then we state that $M \equiv \underline{0}$ in finitely many steps, i.e. $\exists s : M_s \equiv \underline{0}$.

  1. If $M_{y_0} \equiv \underline{0}$ then y_0 is an appropriate choice for s.
  2. Now let us suppose that $M_{y_0} \neq \underline{0}$.

     Since the sequence $M_{y_i}[l_1]$ is constant at component l_1, and properties (1) and (2) hold also for this subsequence, thus, we can find another infinite subsequence $\{M_{v_j}\}$ of $\{M_{y_i}\}$ such that $\{M_{v_j}\}$ is constant at some component $l_2 \neq l_1$ using Steps 1-4 with $\{M_{y_i}\}$ as $\{M_i\}$ and $\{M_{v_j}\}$ as $\{M_{y_i}\}$. As a consequence, the infinite subsequence $\{M_{v_j}\}$ is constant at both components l_1 and l_2 (with $l_1 \neq l_2$).

Since the number of components is equal to n and n is finite, in the same way we can construct an infinite subsequence M_{x_i} such that $M_{x_0}[l_k] = M_{x_i}[l_k]$ for all i, k, and x_0 is finite. However, according to properties (1) and (2) it is possible only if $M_{x_i} \equiv \underline{0}$ for all i. Then choosing $s = x_0$ the theorem is proved. ∎

Then, we claim that mapping $\mathcal{F}()$ is a proper abstraction in the sense that the derived P/T net *without permission places* simulates the original GTS . In other terms, whenever a rewriting step is executed in the GTS on an instance graph, then the corresponding transition can always be fired in the corresponding marking in the P/T net, furthermore, the result marking is an abstraction of the result graph.

Theorem 6.9 (Cardinality P/T net simulates GTS) Let $GTS = (R, TG)$ be a graph transformation system and $PN = (P, T, E, w)$ be a cardinality P/T net derived by the mapping $\mathcal{F}(GTS)$. Furthermore, let G, H be instance graphs typed over TG. Then PN simulates GTS, formally

$$\forall G, H, r, o : (G \overset{r,o}{\Longrightarrow} H) \Rightarrow (M_G \overset{t_r}{\Longrightarrow} M_H),$$

where $\mathcal{F}(G) = M_G$, $\mathcal{F}(H) = M_H$, and $\mathcal{F}(r) = t_r$.

PROOF We need to prove that the following diagram commutes for any match o of rule r with the corresponding transition $t_r = \mathcal{F}(r)$. Our notation used in the proof is summarized by the diagram below.

$$\begin{array}{ccc} M_G & \xrightarrow{t_r = \mathcal{F}(r)} & M_H \\ {\scriptstyle\mathcal{F}}\uparrow & & {\scriptstyle\mathcal{F}}\uparrow \\ G & \xrightarrow{r,o} & H \end{array}$$

Essentially, the proof can be divided into showing that

Step 1: when r is enabled by match o in a graph G of the GTS then t_r is also enabled by the corresponding marking M_G, and

Step 2: the result marking M_H when firing transition t_r is equal to the abstraction $\mathcal{F}(H)$ of the result graph H of the GT step that executes r.

1. Proof of Step 1:

   - We assume by contradiction that r is enabled by o in G but t_r is disabled in M_G.
   - Since o is an injective match that enables r, there is a subgraph G_o of G which is isomorphic with LHS.
   - As a consequence, there are at least as many objects of a certain type y in G as in the LHS, thus $card(LHS, y) \leq card(G, y)$ for all graph object y in TG.
   - By definition of $\mathcal{F}()$, $w(p, t_r) = card(LHS, y)$ for all $p = \mathcal{F}(y)$.

- However, since t_r is disabled (according to our indirect assumption) then for at least one input place p_1 (with $(p_1, t_r) \in E$) we have $M_G(p_1) < w(p_1, t_r)$ due to the enabledness condition of P/T nets.
- For this place p_1, we have $M_G(p_1) < w(p_1, t_r) = card(LHS, y_1) \leq card(G, y_1) = M_G(p_1)$ for some y_1 with $\mathcal{F}(y_1) = p_1$ which is a contradiction.

2. Proof of Step 2:

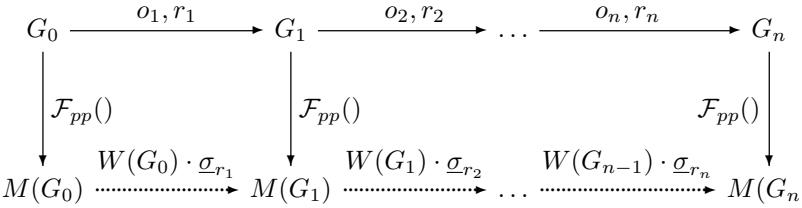
- We now assume that r is enabled in GTS and t_r is enabled in the corresponding PN.
- Since we follow the double pushout approach, exactly those elements are removed from an instance model G when applying a GT rule r which are matched in $LHS \backslash K$ (as potential dangling edges prohibit the application of the rule).
- Similarly, an isomorphic image of the RHS is created as a result of the rule application implying the creation of elements corresponding to $RHS \setminus K$.
- When counting only the number of elements, we can first "remove" and then "add" the image of the context graph K. As a consequence, for all type y in TG, $card(H, y) = card(G, y) - card(LHS, y) + card(RHS, y)$.
- Due to the definition of $\mathcal{F}()$, for all p_y we have $M_H(p) = M_G(p_y) - w(p_y, t_r) + w(t_r, p_y)$, which exactly corresponds to the definition of firing a transition of a P/T net. ∎

Finally, as a termination "oracle", we solve quadratic inequalities based on the incidence matrix of the P/T net with variables as defined in Sec. 6.2.1-6.2.2. If there are no solutions for the inequality for any evaluation of variables in the incidence matrix, we state that the original GTS is terminating.

Theorem 6.10 (Termination) Let $W(\underline{v})$ be the incidence matrix of a cardinality P/T net $PN = \mathcal{F}_{pp}(GTS)$ derived as the abstraction of a GTS.

If $\exists \underline{\sigma} \exists \underline{v} \quad W(\underline{v}) \cdot \underline{\sigma} \geq \underline{0}$ has no solutions with $\underline{v} \geq 1, \underline{\sigma} \geq \underline{0}, \underline{\sigma} \neq \underline{0}$ (thus $\forall \underline{\sigma} \forall \underline{v} \quad \exists k : (W(\underline{v}) \cdot \underline{\sigma})[k] < 0$), then GTS is terminating.

PROOF 1. Let $GT_{seq} = G_0 \stackrel{r_1}{\Longrightarrow} G_1 \stackrel{r_2}{\Longrightarrow} \ldots$ be a transformation sequence (i.e. a sequence of GT steps) as shown in the diagram below.



This sequence implies a sequence of incidence matrices $\{W(G_i)\}$ depending on G_i. In other words, we can calculate the $W(G_i)$ for each GT step $G_i \stackrel{r_{i+1}}{\Longrightarrow} G_{i+1}$ simulating the exact change of tokens at the permission places according to the abstraction $\mathcal{F}_{pp}()$ (like in the case of the initial graph).

2. For each GT step $G_{j-1} \overset{r',o'}{\Longrightarrow} G_j$, we examine that if the occurrences of permission patterns have changed in a certain way for a NAC, how the number of tokens would have changed according to the derived incidence matrix $W(\underline{v})$ of the cardinality P/T net.

   a) *Rule application r', o' generates new occurrences for the permission pattern pp^i_r of a NAC N^i of rule r.* In such a case, the corresponding element $w(t_{r'}, p_{r_{N^i}})$ in the incidence matrix W is a variable v' regardless of o' due to the construction of W.
   Otherwise there are two further possibilities:

   b) *Rule application r', o' disables one or more occurrence of the permission pattern pp^i_r of a NAC N^i of rule r.* In this case, if $r = r'$ then the corresponding element $w(t_{r'}, p_{r_{N^i}})$ in the incidence matrix W is either -1 or 0, If the structure of rule r' guarantees that everything included in the NAC N^i exists or it is created by the RHS, then $w(t_{r'}, p_{r_{N^i}}) = -1$ regardless of match o', otherwise it is 0.
   Finally, if $r \neq r'$ then $w(t_{r'}, p_{r_{N^i}}) = 0$ regardless of match o'.

   c) *Rule application r', o' does not influence the occurrence of the permission pattern pp^i_r of a NAC N^i of rule r.* In this case, the corresponding element $w(t_{r'}, p_{r_{N^i}})$ in the incidence matrix W is either 0 or a variable v'.

   Theorem 6.9 guarantees that we exactly know in other locations of W (belonging to type places) how the number of tokens changes at a GT step.

   As a summary, we notice that the token flow defined by the derived incidence matrix overapproximates the one derived by stepwise abstraction from the GTS.

3. Let $\underline{\sigma}_{r_i}$ be a $|T|$–dimensional unit vector having exactly one non–zero element 1 at the corresponding transition of rule r_i.

$$\begin{aligned}
M(G_j) &= M(G_i) + W(G_i) \cdot \underline{\sigma}_{r_i} + \cdots + W(G_{j-1}) \cdot \underline{\sigma}_{r_{j-1}} \\
M(G_j) - M(G_i) &= W(G_i) \cdot \underline{\sigma}_{r_i} + \cdots + W(G_{j-1}) \cdot \underline{\sigma}_{r_{j-1}} \\
&\leq W(\underline{v} := \underline{c}_i) \cdot \underline{\sigma}_{r_i} + \cdots + W(\underline{v} := \underline{c}_{j-1}) \cdot \underline{\sigma}_{r_{j-1}} \\
&\quad \text{by substituting some non-negative constants } c_i, \ldots, c_j \text{ into } v_i, \ldots, v_j \\
&\leq W(\underline{c}) \cdot \sum_{k=1}^{j-1} \underline{\sigma}_k \text{ where } c[l] = \max\{c_i[l]\} \\
&= W(\underline{c}) \cdot \underline{\sigma}
\end{aligned}$$

   As a summary, we have $M(G_j) - M(G_i) \leq W(\underline{c}) \cdot \underline{\sigma}$. However due to our assumptions, there exists $k : M(G_j)[k] - M(G_i)[k] \leq (W(\underline{c}) \cdot \underline{\sigma})[k] < 0$.

4. As a consequence, Lemma 6.8 can be applied thus we have $M(G_n) \equiv 0$ in finite steps.

5. Since $M(G_i)$ is derived as a step-wise abstraction from the GT sequence, no GT rules are applicable in G_N (as either some graph elements or some permissions are required by a rule, neither of which are present in G_N). ∎

Tool support. In order to show that the quadratic inequality $W(\underline{v}) \cdot \underline{\sigma} \geq \underline{0}$ has no solutions for proving the termination of GTSs with negative application conditions in our example, we used a symbolic optimization toolkit (GAMS [60]) which supports mixed integer non-linear programming.

6.4 Related Work

Relation of graph transformation and Petri nets. The main idea of this paper is to analyze graph transformation systems via Petri nets. In fact, there is a long tradition concerning the relationship of both areas. The basic observation is that a P/T net is essentially a rewriting system on multisets, which allows to encode the firing of P/T nets as a direct graph transformation in the Double Pushout approach using discrete graphs and empty interfaces for the productions only (see [36]). Taking into account general graphs and nonempty interfaces graph transformation systems are closer to some generalizations of Petri nets, like contextual nets. This relationship has been used in [17] to model concurrent computations of graph grammars.

Vice versa the existence of powerful analysis techniques for P/T nets motivates to simulate graph transformation by P/T nets [18], which allows to conclude correctness properties of graph grammars from properties of corresponding P/T nets. The main novelty of this paper wrt. [18] (and subsequent papers of the authors) is that (i) we take into account also negative application conditions of graph transformations and (ii) the size of the derived P/T is dependent on the type graph and not to the instance graph. The price we have to pay for a more efficient termination analysis is that our P/T net can be too abstract to verify all the safety properties investigated in [18].

Termination of graph transformation systems. Termination of graph transformation systems is undecidable in general [112], but several approaches have been considered to restrict a graph transformation system such that termination can be shown. The classical approach of proving termination is to construct a monotone function that measures graph properties, and to show that the value of such a function decreases with every rule application. Concrete criteria such as the number of nodes and edges of certain types have been considered by Aßman in [12]. However, he sticks to these concrete criteria, while Bottoni et.al. [26] developed a general approach to termination based on measurement functions.

With respect to termination for graph transformation systems, the current work generalizes and formalizes the work begun at [41]. This, in fact, is an extension of the layering conditions for deleting grammars proposed in [27], which were used for parsing. A main advantage of our approach with respect to the termination requirements of this parsing algorithm is that we do not require to partition the rules (and the alphabet) into layers.

As pointed out already in the introduction, we have presented termination criteria for graph transformation systems in [K10], which allow to prove termination of several practical relevant model transformations. However these criteria are not applicable to model transformations where rules are causally dependent on themselves (e.g. transitive closure) like our motivating example. Since each layer of [K10] can be treated separately by our current techniques, furthermore, the termination criteria proposed in [K10] imposes a special structure on the derived incidence matrix of the P/T net, it is possible to show that our termination analysis technique based on P/T nets subsumes our former results in [K10].

6.5 Conclusion

In this paper, we have presented a termination analysis technique for model transformations expressed as graph transformation systems using an abstraction into Petri nets. This way, the termination problem of (a special class of) graph transformation systems can be proved by its Petri net abstraction using algebraic techniques. Since the termination of graph transformation systems is undecidable in general, our approach yields a sufficient criterion: either it proves that a GTS is terminating, or gives a “do not know” answer.

We believe that our results can also be useful for proving the termination of QVT-based model transformations, which also uses a very limited set of control structure. For instance, triple graph grammars (TGG) [124] provide a declarative means to specify model transformations, and show a strong conceptual correspondence with bidirectional QVT mappings. Moreover, a pair of traditional (operational) graph transformations can be easily derived for each TGG rule, and then our termination criteria become directly applicable.

Although not mentioned explicitly, the termination criteria presented can also be used for graph transformation with node type inheritance, since a flattening to graph transformation without inheritance is available in [19]. Thus, the termination analysis can always be done and need not be translated back.

Chapter 7

Conclusions

In this last main chapter, I summarize my scientific contributions, and provide a brief overview on the practical relevance and utilization of these results.

7.1 Summary of Scientific Results

7.1.1 Specification Techniques for Model Transformations

Contribution 1 I proposed a general, formal specification language for defining intra-model (endogeneous) and inter-model (exogeneous) model transformations [B3, B6, J5, J11, J14, K1, K6, K8, K11, K25, K27].

1/1 **Graph Patterns as Query Language.** I introduced graph patterns with general recursion and arbitrary depth of negation as a constraint and query language in the context of model transformations.

1/2 **A Hybrid Model Transformations Language.** I proposed the combination of abstract state machines and graph transformation to serve as a hybrid formal language for precisely specifying model transformations.

1/3 **Generic Transformations.** I proposed *generic, higher-order transformations* to provide a compact description for various transformation problems.

The core graph pattern formalism and the hybrid language of the VIATRA2 framework was used by Gergely Varró and Ákos Horváth to support efficient matching of recursive patterns and by András Balogh to improve pattern composition and code generation in their PhD dissertations. Generic and meta-transformations were developed in collaboration with Prof. András Pataricza, were my contribution is primarily related to generic (higher-order) transformations.

Practical relevance. The hybrid formal language combining the paradigm of abstract state machines and graph transformation offers a precise, well-founded yet easy to understand notation for capturing model transformations within and between modeling languages. It has become the transformation language of the VIATRA2 framework, and used in various industrial

and research projects at our research group including collaborative European projects such as DECOS, DIANA, SENSORIA, SecureChange or MOGENTES.

7.1.2 Design Techniques for Model Transformations

Contribution 2 I defined the concepts of *model transformation by example (MTBE)* [B2, J1, K22, K23], which is an iterative and semi-automated approach to derive model transformation rules from a prototypical set of interrelated source and target models, which describe critical cases of the model transformation problem in a purely declarative way.

2/1 **Iterative process for MTBE.** I defined a semi-automated, iterative process for MTBE,which includes (1) the manual creation of prototype mapping models, (2) the automated derivation of transformation rules, (3) the manual refinement of transformation rules by transformation engineer, and (4) the automated execution of transformation rules.

2/2 **Novel MTBE concepts: context and connectivity analysis.** I defined the concepts of context analysis and connectivity analysis for both source and target models in order to identify the required contextual model elements enabling the derivation of model transformation rules.

2/3 **Automation by inductive logic programming.** I proposed the use of inductive logic programming (ILP) to automatically derive model transformation rules for the MTBE approach.

Practical relevance. A main advantage of the MTBE approach is that transformation designers use the concepts of the source and target modeling languages for the specification of the transformation, while the implementation, i.e. the actual model transformation rules are generated (semi-)automatically. In our context, (semi-)automatic rule generation means that transformation designers give hints how source and target models can potentially be interconnected in the form of a mapping metamodel. Then the actual contextual conditions used in the transformation rules are derived automatically based upon the prototypical source and target model pairs.

Model transformation by example has become a hot research topic in the recent years in the MDD community. As many as seven independent groups has started research in that direction in order to propose alternate approaches like [47, 57, 61, 83, 117, 129, 130] yielding more than 90 citations to related papers [J1, K22, K23].

7.1.3 Efficient Execution Techniques for Model Transformations

Contribution 3 I developed efficient execution strategies and concepts for model transformations such as model-specific search plans, incremental model transformations, and compiled transformation plugins [J2–J4,J9,J11,J17–J19,J19–J21,K2–K7,K18–K20,K25,K27].

3/1 **Model-specific search plans.** I proposed the concepts of model-specific search plans, which maintain instance-level statistics to exploit the actual structure of the model for

improving the performance of local-search based execution of graph transformations rules.

3/2 **Incremental graph transformations.** I developed the concept of incremental graph transformation which explicitly store matches of graph transformation rules, and incrementally update these match sets upon changes of the underlying model graph.

3/3 **Compiled model transformation plugins.** I defined a unified architecture for compiled model transformation plugins which allows to embed model transformation rules into native target environments.

This line of research was carried out in close collaboration with several PhD students including István Ráth, Ákos Horváth, Gábor Bergmann (under my formal supervision), András Balogh and Gergely Varró (with my informal co-tutoring). My own contribution lies in proposing *general, high-level strategies and concepts for efficient model transformations*, which were then refined and elaborated into detailed algorithms by these PhD students.

Practical relevance. These results have been implemented as part of the VIATRA2 model transformation framework. Our benchmark investigations [J2, K3, K6] indicated that our model transformations are scalable for models with several million model elements.

Incremental model transformations have been exploited in different ways. *Change-driven model transformations* produce or consume a persisted change model capturing the modifications of a transaction carried out on the source or the target model [J4, K20]. Furthermore, incremental transformation has also been used for *model-driven design-space exploration* [J7, J8, K14, K15]. Finally, incremental model synchronization between abstract and concrete syntax of domain-specific models [J11] and tool integration [B1] has also been carried out on that basis.

7.1.4 Termination Analysis for Model Transformations

Contribution 4 I elaborated a formal analysis technique for proving the termination of model transformations formally captured by graph transformation systems. The approach first derives a Place/Transition (P/T) net abstraction of graph transformation systems, and then calculates the fulfillment of a sufficient termination criteria by algebraic techniques [J7, J8, J22, K10, K14, K15, K26].

4/1 **Petri net abstration of graph transformation systems.** I defined a mapping from graph transformation sytems with negative application conditions and numerical costs of rules to cardinality P/T nets by abstracting from the graph structure.

4/2 **Proof of simulation.** I proved that target cardinality P/T net simulates the source graph transformation system, which guarantees that each run of the source formalism has an equivalent run in the target formalism.

4/3 **Sufficient termination criteria.** I defined a sufficient termination criteria for graph transformation systems based upon the algebraic analysis of the P/T net abstraction.

Practical relevance. While this contribution primarily of theoretical relevance, the Petri net abstraction technique was recently used as a search strategy for solving design space exploration problems [J7, J8, K14, K15]. Furthermore, the same abstraction was extended by Szilvia Varró-Gyapay for the simultaneous optimization and verification of graph transformation systems in [J22].

7.2 Utilization of Scientific Results

Finally, a brief overview is provided on the utilization of the novel scientific results of the thesis. First, a brief overview is given to the VIATRA2 transformation framework, which is an Eclipse based model transformation tool. Then some main usage of the VIATRA2 framework will be provided in the field of service-oriented applications and critical embedded systems.

7.2.1 The VIATRA2 Model Transformation Framework

The main objective of the VIATRA2 (VIsual Automated model TRAnsformations) framework is to provide a general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains.

VIATRA2 primarily aims at designing model transformations to support the precise model-based systems development with the help of invisible formal methods. Invisible formal methods are hidden by automated model transformations projecting system models into various mathematical domains (and, preferably, vice versa). The VIATRA2 model transformation framework is available as an official (open source) Eclipse Generative Modeling Tools (GMT) subproject [4].

VIATRA2 complements other model transformation tools in providing

- a model space for hierarchical and uniform representation of large models and metamodels
- transformation language with both declarative and imperative features based upon popular formal mathematical techniques of graph transformation (GT) and abstract state machines (ASM)
- a high performance transformation engine supporting (1) incremental model transformations, (2) event-driven live transformations where complex model changes may trigger execution of transformations, and (3) handling well over 1,000,000 model elements
- with main target application domains in model transformations for model-based tool integration and model-based analysis

I am the principal investigator of the VIATRA2 framework, which has been developed collaboratively by a powerful team of young researchers, PhD students and MSc students working at the Fault Tolerant Systems Research Group since 2004 in collaboration with employees of OptXware Research and Development Ltd including

- *Chief technologists*: András Balogh, István Ráth
- *Graph transformation and pattern matching experts*: Gergely Varró, Ákos Horváth, Gábor Bergmann;
- *Model space experts*: Zoltán Balogh, András Ökrös, András Schmidt, Balázs Grill;

- *Editors and visualization*: Dávid Vágó, Zoltán Ujhelyi, Ábel Hegedüs

Many insightful comments and ideas from Prof. András Pataricza were also very useful during the development and highly appreciated hereby. I am also indepted to many more researchers and students, who were active users of the VIATRA2 framework, and provided valuable feedback to the development team.

End users. The VIATRA2 framework currently served as the underlying model transformation technology of many European projects in the field of dependable embedded systems and service-oriented applications. In this way, academic and industrial partners in these projects became the first end users of the framework, and provided noticeable international visibility to VIATRA2 . Regular usage of the framework has been reported at ARCS and TU Vienna (Austria), University of Leicester (UK), LMU Munich, TU Kaiserslautern (Germany), University of Trento, University of Pisa (Italy), Georgia University of Technology (USA) and University of Waterloo (Canada).

The VIATRA2 framework also serves as the foundation of an industrial design toolkit for automotive systems developed at OptXware Ltd. for the AUTOSAR architecture. For instance, the validity of design constraints are detected incremental by graph pattern and potential violations are reported immediately to the systems engineers.

7.2.2 Model Transformations for Service-Oriented Applications

The SENSORIA European project developed a comprehensive, model-driven approach for service engineering including (1) novel languages for service modeling, (2) qualitative and quantitative techniques for service analysis, (3) automated mechanisms for model-driven service deployment and (4) transformations for legacy service re-engineering. Model transformation served as a key technology for model-driven service engineering by bridging different languages and tools in the context of service-oriented applications. Various model transformations were developed in the scope of the project:

- **Automated formal analysis of BPEL processes.** The consistency of business processes captured using the standard BPEL notation [104] were formally analyzed by the SAL model checker [21], which exhaustively investigates all potential execution paths of a dynamic behavioral model to decide if a designated property (requirement) holds or not. SAL models were automatically derived [J10, K16] by a complex VIATRA2 model transformation.

- **Back-annotation of model checking results to BPEL processes.** As the reverse transformation problem, back-annotation of model checking results retrieved by the SAL model checker to the BPEL model of service engineers was carried out [K13] by a mapping between traces captured by change-driven transformations [K20].

- **Model-driven performability analysis.** Performability is a non-functional system-level parameter, which aims to assess the cost of using fault-tolerant techniques in terms of performance. We developed a model-driven performability analysis approach [K12] by mapping UML-based service models to formal process model for the PEPA framework [66]. The system level performability model was assembled from a library of core performability components driven by the UML component diagrams of the entire system.

- **Model-driven service deployment.** The derivation of configuration descriptors required for service deployment was automated by a chain of generic model transformations. This

approach was successfully adapted to a number of standard service platforms as reported in [B4, J5, K17].

These results were achieved in collaboration with László Gönczy, Ábel Hegedüs, Gábor Bergmann, István Ráth, András Kövi, Máté Kovács, Zsolt Déri, Tibor Somodi and Tibor Bende under my scientific supervision.

7.2.3 Model Transformations for Critical Embedded Systems

The VIATRA2 model transformation framework has been intensively used for providing tool support for developing and verifying critical embedded systems in the scope of the DECOS, DIANA, MOGENTES, SENSORIA and SecureChange European research projects, and also as part of the INDEXYS project within the industry-driven ARTEMIS platform.

- **Model-driven tool integration.** Model transformations served a key role in tool integration scenarios to bridge a variety of industrial off-the-shelf development tools where the tool integration scenarios were driven by the underlying development process [73,B1,J6].
- **Model-driven development tools** Interactive, user-guided model transformations serve as the foundation for model-driven tools in the automotive and avionics domain aiming to support the development of configuration tables and allocation descriptors [B1].
- **Transformations for ontology-based consistency analysis.** Consistency analysis of domain-specific modeling languages and design models was carried out by following an ontology-based approach. Here model transformations automated the generation of ontological descriptions (e.g. OWL documents) from high-level SysML and UML models.

Key conceptual strategies were outlined by István Majzik, András Pataricza and myself, while additional contributors include Andás Balogh, László Gönczy, Ákos Horváth, Benedek Izsó, Balázs Polgár and István Ráth and Szilvia Varró-Gyapay.

Appendix A

Appendix

A.1 Case Study: A UML-to-Racer Mapping

We present a case study in this section to illustrate the usage of language constructs introduced earlier. We selected a real-life transformation that is used in the European IP DECOS [45]. The transformation maps UML structure models (class diagrams) into ontologies for the Racer reasoner, which can verify the static completeness and consistency of the input model. This transformation is used in the project to validate embedded system models designed using domain-specific metamodels.

The goal of the transformation is the creation of an ontology for a given class structure. The transformation can be described by the following informal rules:

- Each class is mapped to a concept in the Racer model. If the class has a superclass, an implication (representing the inheritance) is created between the two. If the class is a top-level class, an implication is created for the common root class (*TOPC*). The latter is needed to ensure a single inheritance tree.

- Each (binary) association is mapped into the following structure: each association end is mapped to a concept that has two roles; one for each end of the association. This enables a more complex analysis of associations. The concepts created from association ends are inherited from the common root concept *TOPA*.

- Each attribute from the classes is mapped to an attribute for the corresponding concept. An attribute of basic type is mapped to a Racer datatype. This part of the transformation is omitted from the current chapter due to its technical nature.

A.1.1 Graph patterns

We defined several generic graph patterns in Fig. A.1 that can be reused by graph transformation rules. The UML-related ones are grouped into a separate UML pattern library, that can be reused in several transformations, and can be exchanged with libraries designed for other UML versions. The separation of generic, reusable graph patterns for a given modeling language (UML, Racer, etc.) results in an improved maintainability of the transformations.

- Pattern *isUmlClassWithSuperClass* matches UML classes that have superclasses, and *isUmlClassWithoutSuperClass* matches top-level classes.

```
pattern isUmlAssocEndAtClass (AssocEnd, Class) =
{
  Association(Assoc);
  Association.connection(CAA1,Assoc,AssocEndA);
  AssociationEnd(AssocEnd);
  AssociationEnd.type(TAC1,AssocEndA,ClassA);
  Class(Class);
}
pattern isUmlClassWithSuperClass(Class)=
{
  Class(SuperClass);
  Generalization.supertype(SpGG,UmlGen,SuperClass);
  Generalization(UmlGen);
  Generalization.subtype(SbGG,UmlGen,UmlClass);
  Class(UmlClass);
}
pattern isUmlClassWithoutSuperClass(UmlClass) =
{
  Class(UmlClass);
  neg find isUmlClassWithSuperClass(UmlClass);
}
pattern isUmlClassWithAttr(Class, Attr) =
{
  Class(Class);
  Attribute(Attr);
  Classifier.feature(FE,Class,Attr);
}
pattern isUmlAttrOfType(Attr, Type) =
{
 Attribute(Attr);
 DataType(Type);
 StructuralFeature.type(TSC,Attr,Type);
}
pattern isClassWithConcept(Cls,Concept,Ref)=
{
 Class(Cls);
 class2concept.uml(X1,CC,Cls);
 class2concept(CC) in Ref;
 class2concept.racer(X2,CC,Concept);
 concept(Concept);
}
pattern isAssocWithConcept(AscEnd,Conc,Ref)=
{
   AssociationEnd(AscEnd);
   assoc2concept.uml(X1,A2R,AscEnd);
   assoc2concept(A2R) in Ref;
   assoc2concept.racer(X2,A2R,Conc);
   concept(Conc);
}
pattern conceptRole(Role,Domain,Range,Trg) =
{
   concept(Domain);
   role.domain(X1,Role,Domain);
   role(Role) in Trg;
   role.range(X2,Role,Range);
   concept(Domain);
}
pattern conceptImplication(Concept,Subject,Trg) =
{
 concept(Concept);
 concept.impl(IM1,Concept,Impl1);
 implication(Impl1) in Trg;
 implication.subject(IM2,Impl1,Subject);
 concept(Subject);
}
```

Figure A.1: Graph patterns of the case study

- Pattern *isUmlAssocWithEndsBetweenClasses* selects associations with their association ends and connecting classes that can be transformed into Racer.
- Pattern *isUmlClassWithAttr* matches class attributes, and *isUmlAttrOfType* matches attribute data types; both are used for attribute transformation.
- Transformation-specific patterns are *isClassWithConcept* that matches transformed classes together with their associated Racer concepts, and *conceptImplication* which is used for the concept hierarchy.

A.1.2 Graph transformation rules

The rules of the UML-to-Racer mapping are listed in Fig. A.2 and A.3.

```
// Mapping each class to a concept
gtrule class2concept(in Cls, in Trg, in Ref) = {
 precondition pattern pre(Cls, Ref) =  {
   Class(Cls);
   find isUmlClass(Cls);
   neg find isClassWithConcept(Cls,CC, Ref);
 }
 postcondition pattern post(Cls,Trg,Ref,Concept) =  {
   concept(Concept) in Trg;
   find isClassWithConcept(Cls,Concept,Ref);
   Class(Cls);
 }
 action {
  rename(Concept,name(Cls));
 }
}
// Mapping top-level classes to an implication
gtrule topClassImplication (in UmlClass, in TopConcept, in Trg, in Ref) = {
 precondition pattern pre(UmlClass,Concept,Ref) =
 {
   Class(UmlClass);
   find isUmlClassWithoutSuperClass(UmlClass);
   find isClassWithConcept(UmlClass,Concept,Ref);
   concept(Concept);
 }
 postcondition find conceptImplication(Concept,TopConcept,Trg);
}
// Matching classes with supertypes to an implication
gtrule classImplication (in UmlClass, in UmlSuperClass, in Trg, in Ref) =
{
 precondition pattern pre(UmlClass, UmlSuperClass,Concept,SuperConcept, Ref) =
 {
   Class(UmlClass);
   find isUmlClassAndSuperClass(UmlClass,UmlSuperClass);
   Class(UmlSuperClass);
   find isClassWithConcept(UmlSuperClass,SuperConcept,Ref);
   concept(SuperConcept);
   find isClassWithConcept(UmlClass,Concept,Ref);
   concept(Concept);
   neg find conceptImplication(Concept,SuperConcept);
 }
 postcondition find conceptImplication(Concept,TopConcept,Trg);
}
```

Figure A.2: Rules for classes in the UML-to-Racer transformation:

Rule *class2concept* (Fig. A.2) maps classes to Racer concepts. Before mapping it ensures that the class is not already mapped (negative condition). It also creates the reference model

```
gtrule assoc2role(in UmlAssocEnd, in Trg, in Ref, in TopConcept) = {
 precondition pattern pre(UmlAssocEnd,UmlClass,ConceptClass,Ref) =  {
  AssociationEnd(UmlAssocEnd);
  neg find isAssocWithConcept(UMLAssocEnd,Conc,Ref);
  find isUmlAssocEndAtClass (AssocEnd, UmlClass);
  Class(UmlClass);
  find isClassWithConcept(UmlClass,ConceptClass);
  concept(ConceptClass);
 }
 postcondition pattern post(UmlAssocEnd,ConceptClass,
   Conc,RoleA1,RoleA2,Ref,Trg,TopConcept) =
 {
// Precondition elements to be preserved
  AssociationEnd(UmlAssocEnd);
  concept(TopConcept);
  concept(ConceptClass);
// Reference model: created
  find isAssocWithConcept(UMLAssocEnd,Conc,Ref);
// Target model
 // Concepts and implications: created
  concept(Conc) in Trg;
  find conceptImplication(Conc,TopConcept,Trg);
 // Roles
  role(RoleA1) in Trg;
  find conceptRole(RoleA1,ConceptClass,Conc,Trg);
  role(RoleA2) in Trg;
  find conceptRole(RoleA2,Conc,ConceptClass,Trg);
  role.inv(I1,RoleA1,RoleA2);
 }
 action
 {
  rename(Conc,"Assoc_"+name(UmlClass)+"_"+name(UmlAssocEnd));
  rename(RoleA1,name(UmlClass)+"_"+name(UmlAssocEnd)+"_to");
  rename(RoleA2,name(UmlClass)+"_"+name(UmlAssocEnd)+"_from");
 }
}
```

Figure A.3: Rules for associations in the UML-to-Racer transformation

elements together with the concept objects. The *find* construct in the postcondition of the rule denotes that the contents of the called pattern are simply copied to the postcondition pattern (and merged appropriately with existing objects like *Class(C)*).

Rules *topClassImplication* and *classImplication* (Fig. A.2) create the implication (inheritance) relations between the concepts. The first connects the top classes (without superclass) to the special root concept *TOPC*, the second one connects the concepts related to child classes to the concepts of their parents.

Rule *assoc2roles* (Fig. A.3) maps associations ends to Racer model elements. The association ends are mapped to concepts, and they are connected to the associated classes by roles. The created concepts are children of the *TOPA* root concept. The action part of the rule sets the names of the newly created model elements for better readability.

The *main* ASM rule is the entry point of the UML-to-Racer transformation (Fig. A.4). It takes the source, reference and target models as as input, creates the top-level concepts for classes and associations in the Racer model, and calls each graph transformation rule in *forall* in the given sequence.

```
// Main rule: Uml, Ref and Racer are model containers
rule main(in UmlM, in RefM, in RacerM) = seq {
  //create top-most concepts
  new(concept(Topc) in RacerM);
  rename(Topc,"TOPC");
  new(concept(Topa) in RacerM);
  rename(Topa,"TOPA");
  //creating concepts representing UML classes
  forall Class below UmlM with apply class2concept(Class,Racer,Ref);
  //creating concepts and roles representing UML association ends
  forall AssocEnd below UmlM with apply assoc2roles(AssocEnd,Racer,Ref,Topa);
  //creating class hierarchy: top-level classes
  forall Class below UmlM with apply topClassImplication(Class,Topc,Racer,Ref);
  //creating class hierarchy: other classes
  forall Class below UmlM, SuperClass below UmlM with
         apply classImplication(Class,SuperClass,Racer,Ref);
 }
}
```

Figure A.4: ASM control structures

A.2 Experimental Evaluation of Incremental Graph Pattern Matching

A.2.1 Checking consistency constraints for AUTOSAR

The benchmark simulates the typical scenario of model validation. The user is working with a large model, the modifications are small and local, but the result of the validation needs to be computed as fast as possible. To emulate this, the benchmark sequence consists of the following sequence of operations:

1. First, the model is *loaded into memory*. In the case of EMF-IncQuery, most of the overhead is expected to be registered in this phase, as the pattern matching cache needs to be constructed. Note however, that this is a *one-time* penalty, meaning that the cache will be maintained incrementally as long as the model is kept in memory. To highlight this effect, we recorded the times for the loading phase separately.

2. Next, in the *first query phase*, the entire matching set of the constraints is queried. This means that a complete validation is performed on the model, looking for all elements for which the constraint is violated.

3. After the first query, *model manipulations* are executed. These operations only affect a small fixed subset of elements, and change the validity of the constraints.

4. Finally, in the *second query phase*, the complete validation is performed again, to check the net effect of the manipulation operations on the model.

In addition to our EMF-IncQuery-based implementation, we created two separate prototypes: a plain Java variant and an OCL variant that uses MDT-OCL [49]. The exact versions of EMF and MDT-OCL were 2.5.0 and 1.2.0 respectively, running on Eclipse Galileo SR1 20090920-1017. We ran the benchmarks on an Intel Core2 E8400-based PC clocked at 3.00GHz with 3.25GBs of RAM on Windows XP SP3 (32 bit), using the Sun JDK version 1.6.0_17 (with a maximum heap size of 1536 MBs). Execution times were recorded using the `java.lang.System` class, while memory usage data has been recorded in separate runs using the `java.lang.Runtime` class (with several garbage collector invocations to minimize

SSG and iSignal validation pattern in model family A

		EMF/Java			MDT-OCL			INCQuery			
Model Elements #	Model size [MB]	Res [s]	iSignal [s]	SSG [s]	Res [s]	iSignal [s]	SSG [s]	Res [s]	iSignal [s]	SSG [s]	Mem OH [MB]
2 373	30	0.06	0.00	0.25	0.13	0.16	3.58	0.17	0.00	0.00	3
4 748	31	0.08	0.00	0.94	0.16	0.17	13.53	0.22	0.00	0.00	6
9 449	32	0.13	0.01	3.67	0.20	0.19	52.48	0.30	0.00	0.00	12
18 850	33	0.22	0.01	14.52	0.30	0.22	210.48	0.45	0.01	0.00	22
37 721	37	0.42	0.01	58.56	0.47	0.27		0.75	0.01	0.01	45
75 692	43	0.78	0.02	239.53	0.86	0.33		1.58	0.01	0.01	92
151 359	55	1.81	0.03		1.84	0.53		3.22	0.02	0.02	187
302 778	81	3.63	0.06		3.64	0.88		6.19	0.02	0.02	373
605 402	135	7.14	0.09		7.48	1.63		12.00	0.02	0.03	746

Channel validation pattern in model family B

		EMF/Java		MDT-OCL		INCQuery		
Model Elements #	Model size [MB]	Res [s]	Channel [s]	Res [s]	Channel [s]	Res [s]	Channel [s]	Mem OH [MB]
2 972	30	0.06	0.00	0.14	0.17	0.19	0.00	2
6 237	31	0.09	0.02	0.16	0.22	0.27	0.00	4
12 708	32	0.16	0.00	0.25	0.31	0.38	0.00	8
24 885	34	0.28	0.03	0.34	0.33	0.89	0.00	14
47 228	38	0.49	0.06	0.53	0.48	1.28	0.00	28
90 586	44	1.13	0.09	1.20	0.80	2.41	0.00	55
180 389	58	1.94	0.19	2.05	1.41	4.56	0.00	111
370 660	91	4.06	0.39	4.08	2.50	9.00	0.00	225
752 172	156	8.09	0.80	8.11	5.00	20.38	0.00	456
1 558 100	295	17.28	1.59	17.39	10.13	40.22	0.00	943

Legend: Res – resource loading time
Mem OH – memory overhead

Figure A.5: Results overview

the transient effects of Java memory management). The data shown in the results correspond to the averages of 10 runs each.

All implementations share the same Java code for model manipulation, thus they differ only in the query phases:

- The EMF-INCQUERY variant uses our API for reading the matching set of the graph patterns corresponding to constraints. These operations are only dependent on the size of the graph pattern and the size of the matching set itself (this is empirically confirmed by the results, see below). To better reflect memory consumption, the RETE nets for three constraints were built in each case.

- The plain Java variant performs model traversal using the generated model API of EMF. This approach is not naive, but intuitively manually optimized based on the constraint itself (but *not* on the actual structure of the model [K4]).

- The OCL variant has been created by systematically mapping the contents of the graph patterns to OCL concepts, to ensure equivalence. We did not perform any OCL-specific optimization.

To ensure the correctness of the Java implementation, we created a set of small test models and verified the results manually. The rest of the implementations have been checked against the Java variant as the reference, by comparing the number of valid and invalid matches found in each round.

A.2.2 Analysis of the AUTOSAR case study

Based on the results (Fig. A.5), we have made the following observations:

1. As expected, query operations with EMF-INCQUERY are nearly instantaneous, they are only measurable for larger models (where the matching set itself is large). In contrast, both Java and OCL variants exhibit a polynomially increasing characteristic, with respect to model size. The optimized Java implementation outperforms OCL, but only by a constant multiplier.

2. Although not shown in Fig. A.5, the times for model manipulation operations were also measured for all variants, and found to be uniformly negligible. This is expected since very few elements are affected by these operations, therefore the update overhead induced by the RETE network is negligible.

3. The major overhead of EMF-INCQUERY is registered in the resource loading times (shown in the *Res* column in Fig. A.5). It is important to note that the loading times for EMF itself is *included* in the values for EMF-INCQUERY. By looking at the values for loading times and their trends, it can be concluded that EMF-INCQUERY exhibits a linear time increase in both benchmark types, with a factor of approximately 2 compared to the pure EMF implementation. MDT-OCL does not cause a significant increase.

4. The memory overhead also grows linearly with the model size, but depends on the complexity of the constraint too. More precisely, it depends on the size of the match sets of patterns and that of some sub-patterns depending on the structure of the constructed RETE network. (Actually, the memory overhead is sub-linear with respect to patterns, due to a varying degree of RETE node-sharing.)

It has to be emphasized that in practical operations, the resource loading time increase may not be important as it occurs only once, in an initialization phase during a model editing session. So, as long as there is enough memory, EMF-INCQUERY provides nearly instantaneous query performance, independently of the complexity of the query and the contents of the model. In certain cases, like for the SSG and ISignal benchmarks, EMF-INCQUERY is the only variant where the query can be executed in the acceptable time range for large models above 500000 elements, even when we take the combined times for resource loading and query execution into consideration. The performance advantage is less apparent for simple queries, as indicated by the figures for the Channel benchmark, where the difference remains in the range of a few seconds even for large models.

Overall, EMF-INCQUERY suits application scenarios with complex queries, which are invoked many times, with relatively small model manipulations in-between. Even though the memory consumption overhead is acceptable even for large models on today's PCs, the optimization techniques (based on combining various pattern matching techniques [K4]) previously presented for VIATRA2 apply to EMF-INCQUERY too.

dc_244_11

Bibliography

Related Publications

— Books and Book Chapters —

[B1] A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, Á. Horváth, I. Majzik, A. Pataricza, B. Polgár, I. Ráth, D. Varró, and G. Varró. Workflow-driven tool integration using model transformations. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *LNCS*, pages 224–248. Springer, 2010.

[B2] G. Bergmann, A. Boronat, R. Heckel, P. Torrini, I. Ráth, and D. Varró. Advances in model transformation by graph transformations: Specification, analysis and execution. In M. Wirsing and M. Hölzl, editors, *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *LNCS*. Springer, 2011.

[B3] L. Gönczy, Á. Hegedüs, and D. Varró. Methodologies for model-driven development and deployment: an overview. In M. Wirsing and M. Hölzl, editors, *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *LNCS*. Springer, 2011.

[B4] L. Gönczy and D. Varró. *Design and Deployment of Service Oriented Applications with Non-Functional Requirements*, chapter Design and Deployment of Service Oriented Applications with Non-Functional Requirements, pages 315–340. IGI, New York, 2011.

[B5] L. Grunske, L. Geiger, A. Zündorf, N. Van Eetvelde, P. Van Gorp, and D. Varró. *Model Driven Software Engineering*, chapter Using Graph Transformation for Practical Model Driven Software Engineering, pages 91–118. Springer, 2005.

[B6] A. Pataricza and D. Varró. *Formal Methods in Computing*, chapter Metamodeling and Model Transformations, pages 357–425. Akadémiai Kiadó, 2005.

— Peer-reviewed Journals —

[J1] Z. Balogh and D. Varró. Model transformation by example using inductive logic programming. *Software and Systems Modeling*, 8(3):347–364, 2009. IF: 1.533.

[J2] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. Experimental assessment of combining pattern matching strategies with VIATRA2. *Software Tools for Technology Transfer*, 12 (3-4):211–230, 2010.

[J3] G. Bergmann, I. Ráth, and D. Varró. Parallelization of graph transformation based on incremental pattern matching. *Electronic Communications of EASST*, 18, 2009.

[J4] G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations: Change (in) the rule to rule the change. *Software and Systems Modeling*, 2011. IF: 1.27.

[J5] S. Gilmore, L. Gönczy, N. Koch, P. Mayer, M. Tribastone, and D. Varró. Non-functional properties in the model-driven development of service-oriented systems. *Software and Systems Modeling*, 10(3):287–311, 2011. IF: 1.27.

[J6] L. Gönczy, I. Majzik, Á. Horváth, D. Varró, A. Balogh, Z. Micskei, and A. Pataricza. Tool support for engineering certifiable software. *Electr. Notes Theor. Comput. Sci.*, 238 (4):79–85, 2009.

[J7] Á. Hegedüs, A. Horváth, and D. Varró. Towards guided trajectory exploration of graph transformation systems. *Electronic Communications of the EASST*, 40:1–20, 2011.

[J8] A. Horváth and D. Varró. Dynamic constraint satisfaction problems over models. *Software and Systems Modeling*, 2011. IF: 1.27.

[J9] Á. Horváth, D. Varró, and G. Varró. Generic search plans for matching advanced graph patterns. *Electronic Communications of the EASST*, 6, 2007.

[J10] M. Kovács, L. Gönczy, and D. Varró. Formal analysis of BPEL workflows with compensation by model checking. *Int. Journal of Computer Systems and Engineering*, 23(5), 2008. IF: 0.277.

[J11] I. Ráth, A. Ökrös, and D. Varró. Synchronization of abstract and concrete syntax in domain-specific modeling languages. *Software and Systems Modeling*, 9(4):453–471, 2010. IF: 1.533.

[J12] D. Varró. Towards symbolic analysis of visual modelling languages. *Electronic Notes in Theoretical Computer Science*, 72(3):51–64, 2003.

[J13] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and Systems Modeling*, 3(2):85–113, May 2004.

[J14] D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, October 2007. IF: 0.832.

[J15] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Software and Systems Modeling*, 2(3):187–210, October 2003.

[J16] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002. IF: 0.796.

[J17] G. Varró, K. Friedl, and D. Varró. Graph transformation in relational databases. *Electronic Notes in Theoretical Computer Science*, 127(1):167–180, 2005.

[J18] G. Varró, K. Friedl, and D. Varró. Implementing a graph transformation engine in relational databases. *Software and Systems Modelling*, 5(3):313–341, September 2006.

[J19] G. Varró and D. Varró. Graph transformation with incremental updates. *Electronic Notes in Theoretical Computer Science*, 109:71–83, 2004.

[J20] G. Varró, D. Varró, and K. Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electronic Notes in Theoretical Computer Science*, 152:191–205, 2006.

[J21] G. Varró, D. Varró, and A. Schürr. Incremental graph pattern matching: Data structures and initial experiments. *Electronic Communications of the EASST*, 4, 2006.

[J22] S. Varró-Gyapay and D. Varró. Optimization in graph transformation systems using Petri net based techniques. *Electronic Communications of the EASST*, 2, 2006.

— Conference Papers —

[K1] A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, pages 1280–1287, Dijon, France, April 2006. ACM Press. Acc. rate = 32%.

[K2] A. Balogh, G. Varró, D. Varró, and A. Pataricza. Compiling model transformations to EJB3-specific transformer plugins. In *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, pages 1288–1295, Dijon, France, April 2006. ACM Press. Acc. rate = 32%.

[K3] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *Proc. 4th Int. Conf. on Graph Transformations, ICGT 2008*, volume 5214 of *LNCS*, pages 396–410. Springer, 2008. Acc. rate = 40%.

[K4] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. Efficient model transformations by combining pattern matching strategies. In R. F. Paige, editor, *Theory and Practice of Model Transformations, Second Int. Conf., ICMT 2009. Proceedings*, volume 5563 of *LNCS*, pages 20–34. Springer, 2009. Acc. rate = 22%.

[K5] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. Incremental evaluation of model queries over EMF models: A tutorial on EMF-IncQuery. In R. B. France, J. M. Küster, B. Bordbar, and R. F. Paige, editors, *Proc. ECMFA 2011: Modelling Foundations and Applications - 7th European Conference*, volume 6698 of *LNCS*, pages 389–390. Springer, 2011.

[K6] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over EMF models. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems - 13th Int. Conf., MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *LNCS*, pages 76–90. Springer, 2010. Acc. rate: 21%.

[K7] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró. Incremental pattern matching in the VIATRA model transformation system. In G. Karsai and G. Taentzer, editors, *Proc. Graph and Model Transformations (GRAMOT 2008)*. ACM, 2008.

[K8] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró. A graph query language for EMF models. In J. Cabot and E. Visser, editors, *Proc. Int. Conf. on Model Transformation*, volume 6707 of *LNCS*, pages 167–182. Springer, 2011. Acc. rate = 27%.

[K9] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In J. Richardson, W. Emmerich, and D. Wile, editors, *Proc. ASE 2002: 17th IEEE Int. Conf. on Automated Software Engineering*, pages 267–270, Edinburgh, UK, September 23–27 2002. IEEE Press. Acc. rate = 20%.

[K10] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Cerioli, editor, *Proc. FASE 2005: Int. Conf. on Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 49–63, Edinburgh, UK,, April 2005. Springer. Acc. rate = 22%.

[K11] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, Int. Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.

[K12] L. Gönczy, Z. Déri, and D. Varró. Model transformations for performability analysis of service configurations. In M. R. V. Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008. Reports and Revised Selected Papers*, volume 5421 of *LNCS*, pages 153–166. Springer, 2008.

[K13] Á. Hegedüs, G. Bergmann, I. Ráth, and D. Varró. Back-annotation of simulation traces with change-driven model transformations. In *Proceedings of the Eighth Int. Conf. on Software Engineering and Formal Methods (SEFM 2010)*, pages 145–155, Pisa, 09/2010 2010. IEEE Computer Society. Acc. rate: 22%.

[K14] Á. Hegedüs, A. Horváth, and D. Varró. A model-driven framework for guided design space exploration. In *Proc. ASE 2011: 26th IEEE/ACM Int. Conf. On Automated Software Engineering*, pages 173–182. IEEE Computer Society, November 2011. Acc. rate = 16%, **ACM Distinguished Paper Award**.

[K15] Á. Horváth and D. Varró. CSP(M): Constraint Satisfaction Problem over Models. In A. Schürr and B. Selic, editors, *Model Driven Engineering Languages and Systems, 12th Int. Conf., MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, volume 5795 of *LNCS*, pages 107–121. Springer, 2009. Acc. rate: 18%.

[K16] M. Kovács, L. Gönczy, and D. Varró. Formal modeling of BPEL workflows including fault and compensation handling. In *EFTS '07: Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, page 1, New York, NY, USA, 2007. ACM.

[K17] A. Kövi and D. Varró. An Eclipse-based framework for AIS service configurations. In M. Malek, M. Reitenspieß, and A. P. A. van Moorsel, editors, *Proc. 4th Int. Service Availability Symposium, ISAS 2007, Durham, NH, USA, May 21-22, 2007*, volume 4526 of *LNCS*, pages 110–126. Springer, 2007.

[K18] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live model transformations driven by incremental pattern matching. In A. Vallecillo, J. Gray, and A. Pierantonio, editors,

Proc. First Int. Conf. on the Theory and Practice of Model Transformations (ICMT 2008), volume 5063 of *LNCS*, pages 107–121. Springer, 2008. Acc. rate = 31%.

[K19] I. Ráth, D. Vago, and D. Varró. Design-time simulation of domain-specific models by incremental pattern matching. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008, Herrsching am Ammersee, Germany, 15-19 September 2008, Proceedings*, pages 219–222. IEEE, 2008. Acc. rate = 29%.

[K20] I. Ráth, G. Varró, and D. Varró. Change-driven model transformations. In A. Schürr and B. Selic, editors, *Proc. Model Driven Engineering Languages and Systems, 12th Int. Conf., MODELS 2009*, volume 5795 of *LNCS*, pages 342–356. Springer, 2009. Acc. rate: 18%, **Springer Best Paper Award and ACM Distinguished Paper Award**.

[K21] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. ICGT 2004: Second Int. Conf. on Graph Transformation*, volume 3256 of *LNCS*, pages 226–241, Rome, Italy, 2004. Springer.

[K22] D. Varró. Model transformation by example. In *Proc. Model Driven Engineering Languages and Systems (MODELS 2006)*, volume 4199 of *LNCS*, pages 410–424, Genova, Italy, 2006. Springer. Acc. rate = 29%.

[K23] D. Varró and Z. Balogh. Automating model transformation by example using inductive logic programming. In Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC 2007), Seoul, Korea, March 11-15, 2007*, pages 978–984. ACM Press, 2007. Acc. rate = 32%.

[K24] D. Varró and A. Pataricza. Automated formal verification of model transformations. In J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, editors, *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, pages 63–78. TU München, September 2003.

[K25] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *Proc. UML 2004: 7th Int. Conf. on the Unified Modeling Language*, volume 3273 of *LNCS*, pages 290–304, Lisbon, Portugal, October 10–15 2004. Springer. Acc. rate = 22%.

[K26] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination analysis of model transformations by Petri nets. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Proc. Third Int. Conf. on Graph Transformation (ICGT 2006)*, volume 4178 of *LNCS*, pages 260–274, Natal, Brazil, 2006. Springer. Acc. rate = 45%.

[K27] G. Varró, Á. Horváth, and D. Varró. Recursive graph pattern matching: With magic sets and global search plans. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. Third Int. Workshop and Symposium on Applications of Graph Transormation with Industrial Relevance (AGTIVE 2007)*, volume 5088 of *LNCS*. Springer, 2008.

[K28] G. Varró, A. Schürr, and D. Varró. Benchmarking for graph transformation. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 05)*, pages 79–88, Dallas, Texas, USA, September 2005. IEEE Press. Acc. rate = 31%.

External References

[1] *Prover9: Automated Theorem Prover*. http://www.cs.unm.edu/\~mccune/prover9/.

[2] *StylisStudio*. http://www.stylusstudio.com.

[3] *The Aleph Manual*. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

[4] *VIATRA2 Framework*. An Eclipse GMT Subproject (http://www.eclipse.org/gmt/VIATRA2).

[5] Model transformations in practice workshop, 2005. http://sosym.dcs.kcl.ac.uk/events/mtip/.

[6] H. Ade and M. Denecker. AILP: Abductive inductive logic programming. In *IJCAI*, pages 1201–1209, 1995.

[7] Aeronautical Radio, Incorporated (ARINC). ARINC 653: Avionics Application Standard Software Interface, 2006. Supplement 2: http://www.arinc.com.

[8] Altova:. *MapForce 2006*. http://www.altova.com/features_xml2xml_mapforce.html.

[9] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006*, volume 4066 of *LNCS*, pages 361–375. Springer, 2006.

[10] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Proceedings, Part I*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

[11] U. Assmann. *In [51]*, chapter OPTIMIX: A Tool for Rewriting and Optimizing Programs, pages 307–318. World Scientific, 1999.

[12] U. Aßmann. Graph rewrite systems for program optimization. *ACM TOPLAS*, 22(4): 583–637, 2000.

[13] M. Asztalos, L. Lengyel, and T. Levendovszky. Towards automated, formal verification of model transformations. In *Proc. 3rd International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 15–24. IEEE Computer Society, 2010.

[14] C. Atkinson and T. Kühne. The essence of multilevel metamodelling. In M. Gogolla and C. Kobryn, editors, *Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools*, volume 2185 of *LNCS*, pages 19–33. Springer, 2001.

[15] AUTOSAR Consortium. *The AUTOSAR Standard.* http://www.autosar.org/.

[16] D. Balasubramanian, A. Narayanan, C. P. van Buskirk, and G. Karsai. The graph rewriting and transformation language: Great. *ECEASST*, 1, 2006.

[17] P. Baldan. *Modelling Concurrent Computations: From Contextual Petri Nets to Graph Grammars*. PhD thesis, University of Pisa, 2000.

[18] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In K. G. Larsen and M. Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference*, volume 2154 of *LNCS*, pages 381–395, Aalborg, Denmark, August 20-25 2001. Springer.

[19] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating meta modelling with graph transformation for efficient visual language definition and model manipulation. In *Proc. FASE'04: Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 214–228. Springer, 2004.

[20] D. Batory. The LEAPS algorithm. Technical Report CS-TR-94-28, 1, 1994.

[21] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, 2000.

[22] J. Bézivin. On the unification power of models. *Software and Systems Modelling*, 4(2): 171–188, 2005.

[23] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. In P. Stevens, J. Whittle, and G. Booch, editors, *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, volume 2863 of *LNCS*, pages 175–189, San Francisco, CA, USA, October 20-24 2003. Springer.

[24] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, 16(5):265–275, 2001.

[25] E. Börger and R. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[26] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Termination of high-level replacement units with application to model transformation. *ENTCS*, 127(4), 2005.

[27] P. Bottoni, G. Taentzer, and A. Schürr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *Proc. Visual Languages 2000*, pages 59–60. IEEE Computer Society, 2000.

[28] H. Bunke, T. Glauser, and T.-H. Tran. An efficient implementation of graph grammar based on the RETE-matching algorithm. In *Proc. Graph Grammars and Their Application to Computer Science and Biology*, volume 532 of *LNCS*, pages 174–189, 1991.

[29] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba Real-Time tool suite: model-driven development of safety-critical, real-time systems. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 670–671. ACM, 2005.

[30] J. Cabot and E. Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *J. Syst. Softw.*, 82(9):1459–1478, 2009.

[31] K. Chen, J. Sztipanovits, S. Abdelwahed, and E. K. Jackson. Semantic anchoring with model transformations. In *ECMDA-FA*, pages 115–129, 2005.

[32] K. Chen, J. Sztipanovits, and S. Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In W. Wolf, editor, *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, pages 35–43. ACM, 2005.

[33] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[34] A. Cicchetti, D. di Ruscio, and R. Eramo. Towards propagation of changes by model approximations. In *In Proc. of the 10th International Enterprise Distributed Object Computing Conference Workshops (EDOC 2006)*, page 24. IEEE Computer Society, 2006. Workshop on Models of Enterprise Computing.

[35] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming*. CSLI Publications, Stanford University, 2000.

[36] A. Corradini. Concurrent graph and term graph rewriting. In *Proc. CONCUR'96*, volume 1119 of *LNCS*, pages 438–464. Springer, 1996.

[37] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.

[38] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *In [119]*, chapter Algebraic Approaches to Graph Transformation — Part I: Basic Concepts and Double Pushout Approach, pages 163–245. World Scientific, 1997.

[39] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. The MIT Press, 1993.

[40] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[41] J. de Lara and G. Taentzer. Automated model transformation and its validation with atom3 and agg. In *Proc. DIAGRAMS'2004 (Cambridge, UK)*, volume 2980 of *LNAI*, pages 182–198. Springer, 2004.

[42] J. de Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber, editors, *5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.

[43] J. de Lara and H. Vangheluwe. Translating model simulators to analysis models. In *Proc. FASE 2008*, pages 77–92, 2008.

[44] L. De Raedt and N. Lavrač. Multiple predicate learning in two inductive logig prgramming settings. *Journal on Pure and Applied Logic*, 4(2):227–254, 1996.

[45] DECOS. Dependable Components and Systems. an FP6 Integrated Project. http://www.decos.at.

[46] M. Didonet Del Fabro, J. Bézivin, F. Jouault, and P. Valduriez. Applying generic model management to data mapping. In *Journées Bases de Données Avancés (BDA)*, pages 343–355, 2005.

[47] M. Didonet Del Fabro and P. Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and System Modeling*, 8(3):305–324, 2009.

[48] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. http://eclipse.org/modeling/emf/.

[49] The Eclipse Project. *MDT OCL*. http://www.eclipse.org/modeling/mdt/?project=ocl.

[50] J. Edmonds. Optimum branchings. *Journal Research of the National Bureau of Standards*, 71(B):233–240, 1967.

[51] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

[52] H. Ehrig and C. Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In *Proc. ICGT 2008*, pages 194–210, 2008.

[53] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *In [119]*, chapter Algebraic Approaches to Graph Transformation — Part II: Single pushout approach and comparison with double pushout approach, pages 247–312. World Scientific, 1997.

[54] C. Ermel, M. Rudolf, and G. Taentzer. *In [51]*, chapter The AGG-Approach: Language and Tool Environment, pages 551–603. World Scientific, 1999.

[55] M. Erwig. Toward the automatic derivation of XML transformations. In *1st Int. Workshop on XML Schema and Data Management (XSDM'03)*, volume 2814 of *LNCS*, pages 342–354. Springer, 2003.

[56] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data Knowl. Eng.*, 70(5):448–466, 2011.

[57] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *LNCS*, pages 326–340. Springer, 2008.

[58] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Theory and Application to Graph Transformations (TAGT'98)*, volume 1764 of *LNCS*. Springer, 2000. ISBN 3-540-67203-6.

[59] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.

[60] GAMS: General Algebraic Modeling System. `http://www.gams.com`.

[61] I. García-Magarino, J. J. Gómez-Sanz, and R. Fuentes-Fernández. Model transformation by-example: An algorithm for generating many-to-many transformation rules in several model transformation languages. In *ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 52–66, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02407-8.

[62] R. Geiß, V. Batz, D. Grund, S. Hack, and A. M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. of the 3rd International Conference on Graph Transformation*, volume 4178 of *LNCS*, pages 383–397, Natal, Brazil, September 2006. Springer.

[63] R. Geiß and M. Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, volume 5088 of *LNCS*, pages 568–569. Springer, 2007.

[64] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. ICGT 2002: Firs International Conference on Graph Transformation*, volume 2505 of *LNCS*, pages 90–105, Barcelona, Spain, October 7–12 2002. Springer-Verlag.

[65] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling (SoSyM)*, 8(1), 3 2009.

[66] S. Gilmore and J. Hillston. The PEPA Workbench: A tool to support a process algebra-based approach to performance modelling. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation, Modeling Techniques and Tools, 7th International Conference, Vienna, Austria, May 3-6, 1994, Proceedings*, volume 794 of *LNCS*, pages 353–368. Springer, 1994.

[67] J. Greenyer and E. Kindler. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.

[68] I. Groher, A. Reder, and A. Egyed. Incremental consistency checking of dynamic constraints. In *Fundamental Approaches to Software Engineering (FASE 2009)*, volume 6013 of *LNCS*, pages 203–217. Springer, 2010.

[69] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Proceedings*, pages 157–166, Washington, D.C., USA, 1993.

[70] Y. Gurevich. The sequential ASM thesis. *Bulletin of the European Association for Theoretical Computer Science*, 67:93–124, 1999.

[71] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 321–335, Genova, Italy, 2006.

[72] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. ICGT 2002: First International Conference on Graph Transformation*, volume 2505 of *LNCS*, pages 161–176, Barcelona, Spain, October 7–12 2002. Springer.

[73] W. Herzner, B. Huber, G. Csertán, and A. Balogh. The DECOS tool-chain: Model-based development of distibuted embedded safety-critical real-time systems. In *Proc. of the DECOS/ERCIM Workshop at SAFECOMP 2006*, pages 22–24. ERCIM, 2006.

[74] S. E. Hudson. Incremental attribute evaluation: an algorithm for lazy evaluation in graphs. Technical Report 87-20, University of Arizona, 1987.

[75] G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis, and M. Dal Cin. Quantitative analysis of UML Statechart models of dependable systems. *The Computer Journal*, 45(3):260–277, May 2002.

[76] S. Islam, N. Suri, A. Balogh, G. Csertán, and A. Pataricza. An optimization based design for integrated dependable real-time embedded systems. *Design Autom. for Emb. Sys.*, 13 (4):245–285, 2009.

[77] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[78] F. Jouault and J. Bézivin. KM3: A DSL for Metamodel Specification. In *Proc. FMOODS 2006: Formal Methods For Open Object-Based Distributed Systems*, volume 4037 of *LNCS*, pages 171–185. Springer, 2006.

[79] F. Jouault and I. Kurtev. Transforming models with ATL. In *Model Transformations in Practice Workshop at MODELS 2005*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.

[80] F. Jouault and M. Tisi. Towards incremental execution of ATL transformations. In *Proc. of ICMT'10, 3rd Intl. Conference on Model Transformation*, volume 6142 of *LNCS*, pages 123–137. Springer, 2010.

[81] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Revised Selected Papers*, volume 3599 of *LNCS*, pages 62–76, Linköping, Sweden, 2005. Springer.

[82] G. Karsai and A. Narayanan. On the correctness of model transformations in the development of embedded systems. In F. Kordon and O. Sokolsky, editors, *Composition of Embedded Systems. Scientific and Industrial Issues, 13th Monterey Workshop 2006, Paris, France, October 16-18, 2006, Revised Selected Papers*, volume 4888 of *LNCS*, pages 1–18. Springer, 2007.

[83] M. Kessentini, H. A. Sahraoui, and M. Boukadoum. Model transformation as an optimization problem. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *LNCS*, pages 159–173. Springer, 2008.

[84] T. Klein, U. Nickel, J. Niere, and A. Zündorf. From UML to Java and back again. Technical report, University of Paderborn, 2000.

[85] A. Königs and A. Schürr. MDI - a rule-based multi-document and tool integration approach. *Software and Systems Modelling*, 5(4):349–368, 2006.

[86] J. Larrosa and G. Valiente. Graph pattern matching using constraint satisfaction. In H. Ehrig and G. Taentzer, editors, *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, Berlin, Germany, March 25–27 2000.

[87] M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Revised Selected Papers*, volume 3844 of *LNCS*, pages 139–150. Springer, 2006.

[88] S. Lechner and M. Schrefl. Defining web schema transformers by example. In V. Marik, W. Retschitzegger, and O. Stepankova, editors, *DEXA*, volume 2736 of *LNCS*, pages 46–56. Springer, 2003.

[89] L. Lengyel, T. Levendovszky, and H. Charaf. Validated model transformation-driven software development. *IJCAT*, 31(1/2):106–119, 2008.

[90] T. Levendovszky, L. Lengyel, G. Mezei, and H. Charaf. A systematic approach to metamodeling environments and model transformation systems in VMTS. In *Electr. Notes Theor. Comput. Sci.*, volume 127, pages 65–75. Elsevier, 2005. Proc. GraBaTs 2004: International Workshop on Graph Based Tools.

[91] B. G. T. Lowden and J. Robinson. Constructing inter-relational rules for semantic query optimisation. In A. Hameurlain, R. Cicchetti, and R. Traunmüller, editors, *Proceedings of Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, September 2-6,*, volume 2453 of *LNCS*, pages 587–596. Springer, 2002.

[92] Z. Á. Mann, A. Orbán, and P. Arató. Finding optimal hardware/software partitions. *Formal Methods in System Design*, 31:241–273, 2007.

[93] A. Matzner, M. Minas, and A. Schulte. Efficient graph matching with application to cognitive automation. In M. Nagl and A. Schürr, editors, *Proc. of the 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance*, pages 293–308, Kassel, Germany, October 2007.

[94] S. J. Mellor. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, 2004.

[95] T. Mészáros. *Supporting Model Animation Methods with Graph Transformation*. PhD thesis, Budapest University of Technology and Economics, 2011.

[96] T. Mészáros and et al. Manual and automated performance optimization of model transformation systems. *Software Tools for Technology Transfer*, 12(3-4):231–243, 2010.

[97] S. P. Miller. Certification issues in model based development. Technical report, Advanced Technology Center, Rockwell Collins, 2006.

[98] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl. Proving the shalls. *STTT*, 8(4-5):303–319, 2006.

[99] D. P. Miranker and B. J. Lofaso. The organization and performance of a TREAT-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3 (1):3–10, 1991.

[100] S. Moyle. Using theory completion to learn a navigation control program. In S. Matwin and C. Sammut, editors, *Proc. Twelfth International Conference on ILP (ILP 2002)*, volume LNAI, pages 182–197. Springer, 2003.

[101] S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19-20:629–679, 1994.

[102] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4): 541–580, 1989.

[103] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, 2000. ACM Press.

[104] OASIS. Web Services Business Process Execution Language Version 2.0 (OASIS Standard), 2007. "http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html".

[105] Object Management Group. *Meta Object Facility Version 2.0*, 2003. http://www.omg.org.

[106] Object Management Group. *UML Semantics Version 2.0*, May 2003. http://www.omg.org.

[107] Object Management Group. *Object Constraint Language Specification (Version 2.0)*, May 2006. http://www.omg.org.

[108] Object Management Group. QVT: MOF 2.0 Query / View / Transformation, 2008. http://www.omg.org/spec/QVT/1.0/.

[109] Object Management Group. *SysML: Systems Modeling Language*, June 2010. http://www.sysml.org.

[110] K. Ono, T. Koyanagi, M. Abe, and M. Hori. XSLT stylesheet generation by example with WYSIWYG editing. In *Proceedings of the 2002 Symposium on Applications and the Internet (SAINT 2002)*, pages 150–161, Washington, DC, USA, 2002. IEEE Computer Society.

[111] A. Pataricza. Model-based dependability analysis, 2008. DSc thesis, Hungarian Academy of Sciences.

[112] D. Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33 (2):201–209, 1998.

[113] J. Poole, D. Chang, D. Tolbert, and D. Mellor. *Common Warehouse Metamodel*. John Wiley & Sons, Inc., 2002.

[114] Racer Systems Gmbh. Racerpro. http://www.racer-systems.com.

[115] A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proc. 2nd International Conference on Graph Transformation (ICGT 2004), Rome, Italy*, volume 3256 of *LNCS*, pages 319–335. Springer, 2004.

[116] A. Repenning and C. Perrone. Programming by example: programming by analogous examples. *Communications of the ACM*, 43(3):90–97, 2000.

[117] R. Robbes and M. Lanza. Example-based program transformation. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *LNCS*, pages 174–188. Springer, 2008.

[118] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model migration with epsilon flock. In *ICMT*, pages 184–198, 2010.

[119] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.

[120] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[121] SAE. *Architecture Analysis and Design Language (AADL)*. http://www.aadl.info/.

[122] E. Schoitsch, E. Althammer, H. Eriksson, J. Vinter, L. Gönczy, A. Pataricza, and G. Csertán. Validation and certification of safety-critical embedded systems - The DECOS Test Bench. In J. Górski, editor, *Computer Safety, Reliability, and Security, 25th International Conference, SAFECOMP 2006, Gdansk, Poland, September 27-29, 2006, Proceedings*, volume 4166 of *LNCS*, pages 372–385. Springer, 2006.

[123] A. Schürr. Introduction to PROGRES, an attributed graph grammar based specification language. In M. Nagl, editor, *Graph–Theoretic Concepts in Computer Science*, volume 411 of *LNCS*, pages 151–165, Berlin, 1990. Springer.

[124] A. Schürr. Specification of graph translators with triple graph grammars. In B. Tinhofer, editor, *Proc. WG94: International Workshop on Graph-Theoretic Concepts in Computer Science*, number 903 in LNCS, pages 151–163. Springer, 1994.

[125] A. Schürr, A. J. Winter, and A. Zündorf. *In [51]*, chapter The PROGRES Approach: Language and Environment, pages 487–550. World Scientific, 1999.

[126] S. Shekhar, B. Hamidzadeh, A. Kohli, and M. Coyle. Learning transformation rules for semantic query optimization: A data-driven approach. *IEEE Trans. Knowl. Data Eng.*, 5 (6):950–964, 1993.

[127] G. Simon, G. Karsai, G. Biswas, S. Abdelwahed, N. Mahadevan, T. Szemethy, G. Péceli, and T. Kovácsházy. Model-based fault-adaptive control of complex dynamic systems. In *Proceedings of the 20th IEEE Instrumentation and Measurement Technology Conference, IMTC/2003*, pages 176–181. IEEE, 2003.

[128] D. D. Straube and M. T. Özsu. Query optimization and execution plan generation in object-oriented data management systems. *Knowledge and Data Engineering*, 7(2):210–227, 1995.

[129] M. Strommer, M. Murzek, and M. Wimmer. Applying Model Transformation By-Example on Business Process Modeling languages. In *Proc. 3rd International Workshop on Foundations and Practices of UML (ER 2007)*, volume 4802 of *LNCS*, pages 116–125. Springer, 2007.

[130] Y. Sun, J. White, and J. Gray. Model transformation by demonstration. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 712–726, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04424-3.

[131] Sun Microsystems. *Enterprise Java Beans 3.0*.

[132] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, 30(4): 110–111, 1997.

[133] A. Tiwari, N. Shankar, and J. M. Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, 2003.

[134] L. Tratt. The MT model transformation language. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France*, pages 1296–1303. ACM, 2006.

[135] G. Varró. *Advanced Techniques for the Implementation of Model Transformation Systems*. PhD thesis, Budapest University of Technology and Economics, 2008.

[136] G. Varró. Implementing an EJB3-specific graph transformation plugin by using database independent queries. *Electr. Notes Theor. Comput. Sci.*, 211:121–132, 2008.

[137] A. Vizhanyo, A. Agrawal, and F. Shi. Towards generation of efficient transformations. In G. Karsai and E. Visser, editors, *Proc. of 3rd Int. Conf. on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *LNCS*, pages 298–316, Vancouver, Canada, October 2004. Springer-Verlag.

[138] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards model transformation generation by-example. In *Proc. of HICSS-40 Hawaii International Conference on System Sciences*, page 285, Hawaii, USA., January 2007. IEEE Computer Society.

[139] I. Wright and J. Marshall. The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. *International Journal of Intelligent Games and Simulation*, 2 (1):36–48, February 2003.

[140] L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 485–496, 2001.

[141] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In D. S. Kerr, editor, *VLDB*, pages 1–24. ACM, 1975.

[142] A. Zündorf. Graph pattern-matching in PROGRES. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*, pages 454–468. Springer-Verlag, 1996.