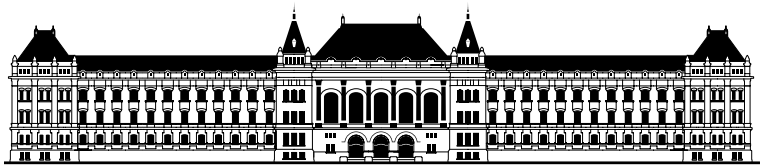


Budapest University of Technology and Economics
Department of Computer Science and Information Theory

Advanced Techniques for the Implementation of Model Transformation Systems

PhD Thesis

Gergely Varró

MSc in Technical Informatics

Supervisors:

Dr. Katalin Friedl, PhD
associate professor

Dr. Dániel Varró, PhD
assistant professor

Prof. Dr. rer. nat. Andy Schürr
professor

Budapest, April 2008

Nyilatkozat

Alulírott, *Varró Gergely György*, kijelentem, hogy ezt a doktori értekezést magam készítettem, és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2008. április 16.

A dolgozat bírálatai és a védésről készült jegyzőkönyv a későbbiekben a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Dékáni Hivatalában elérhetők.

Advanced Techniques for the Implementation of Model Transformation Systems

Gergely Varró

PhD thesis summary

Abstract

When developing software applications in a model-driven way by the MDD paradigm, the high-level system models designed by software engineers are automatically converted to platform-specific representations (such as J2EE, .NET, or CORBA) and later to program code by *model transformations*. In the current thesis, I propose advanced support for executing complex model transformations. I also analyze the performance and the tool integration capabilities of model transformation systems.

In software engineering, the leading trend of Model-Driven Development (MDD) aims at creating system models on various abstraction levels, and automatically transforming these models into each other. In this process, a large number of modeling languages and tools are involved. Powerful domain-specific modeling environments frequently provide rich support for developing editors, and code generators, but the design of model transformations are usually not supported properly in these industrial tools. This thesis primarily focuses on to provide advanced support for executing complex model transformations within and between these modeling languages.

The MDD approach requires these transformations to be (i) captured by a high-level specification language, (ii) automatically executed by efficient algorithms and techniques, and (iii) extensively supported by industrial quality tools.

Though model transformations can be appropriately defined by the specification languages of the Query/Views/Transformations (QVT) standard, several performance and tool integration related issues are missing from both the design and the implementation of model transformation algorithms, techniques and tools despite the fact that the declarative and rule-based paradigm of graph transformation already provides a well-defined formal specification framework for implementing model transformations.

In the current thesis, I propose several advanced, graph transformation based techniques for the implementation of model transformation systems by also assessing their performance and analyzing their tool integration capabilities.

Benchmarking framework for graph transformation. I propose a benchmarking framework, which enables quantitative performance analysis and comparison of graph transformation tools and their optimization strategies.

Graph transformation in relational databases. I elaborate a provenly correct method for the implementation of graph transformation built on top of a relational database, and I assess the performance of the approach by using different databases and several parameter and optimization strategy settings.

Adaptive graph transformation. I present an adaptive method for executing model-specific search plans in order to improve the performance of graph transformation in its pattern matching phase.

Incremental graph transformation. I elaborate a notification framework based incremental method for graph pattern matching, which stores partial matchings explicitly in the main memory and updates them incrementally, when notifications about model changes arrive. Additionally, I assess the performance of the approach by comparing it to a traditional graph transformation tool.

Contents

Acknowledgements

Prior to discussing the theoretical and practical details of implementing advanced model transformation systems, I would like to cordially thank all those people who gave significant assistance to me during the recent years.

The research that lead to the contributions of this thesis has been carried out in a very special and unusual configuration of supervision and working environment, which resulted in a fruitful and effective inter-department and international cooperation. Thus, I would like to say many thanks to all those colleagues who participated in forming and maintaining this special alliance.

In this sense, first, I highly acknowledge Katalin Friedl, who undertook the supervision of my research even under these very special circumstances. Her wonderful and always supportive personality had a large role in creating and preserving the protected and cooperative working environment, which helped me a lot to be able to focus on my research related tasks. I am grateful to Kati for the vivid discussions on improving the details of algorithms, for her useful guidelines, thorough proofreading and many hints in the process of writing precise, high quality, technical papers.

Though it would be the place for the acknowledgements, I cannot express in words how grateful I am to my brother, Dániel Varró, who was a brother, a friend, a supervisor, a colleague, and a co-author of many papers at the same time in one person. Five years ago he proposed and convinced me to stay involved in the "family business" and to continue my research on model transformation. He suggested me many of my high-level research goals, of which all are proved to be achievable (as demonstrated by this thesis). Moreover, they really served as a guideline for me to find the lower level research goals and solutions. I should highly acknowledge Dani for the many-hour discussions, which were always useful and really inspiring; for teaching me the technique of writing research papers; for all the know-how I acquired from him to sell my results on international forums and for encouraging me many times not to give up my long-term plans.

In 2004, I was invited by Andy Schürr to spend five months as a visiting researcher at the Real-Time Systems Lab of the Institute of Computer Engineering at the Technical University of Darmstadt with the financial support of the SegraVis training network. I highly acknowledge Andy for teaching me to notice the practice oriented aspects of software engineering problems and to solve them by also taking into account many practical considerations. In this sense, I am very grateful for the idea of creating the benchmarking framework, for guiding the complete process of the related research, for giving me an overview on existing incremental techniques and for suggesting improvements to these algorithms. Additionally, I say many thanks for the vivid, weekly discussions, for the comfortable and unperturbed working environment, which helped me a lot to concentrate only on my research tasks, and finally, on a rather personal background, for the trust to invite me to Darmstadt without knowing me personally.

András Recski is acknowledged for providing the warm, kind, friendly, open and bureaucracy free environment at the Department of Computer Science and Information Theory (Budapest University of Technology and Economics) as the head of the department. Since my presentation skills that proved to be useful in conferences have been improved a lot while teaching Formal Languages practical courses, I say many thanks to Iván Bach for inviting me to be a teaching assistant and to Judit Csima, Gergely Lukácsy, Mátyás Naszódi, Ildikó Schlotter, Zsolt Terék and many others for being such wonderful and helpful colleagues. My applications for many scholarships and other financial support could not be successful without the administrative help of Katalin Czenkiné Boltizár.

Since my work was more closely related to the research areas of the Fault Tolerant Systems Research Group at the Department of Measurement and Information Systems (Budapest University of Technology and Economics), I am very grateful to András Pataricza for patronizing this fruitful inter-department cooperation as the head of the group, and for being my supervisor in the period of my

undergraduate studies. On a more personal level, I say many thanks for his support and trust during the recent years. Though I should acknowledge the help of many colleagues at the Fault Tolerant Systems Research Group, the discussions with András Balogh, István Ráth, Ákos Horváth, and Gábor Bergmann on the development of the VIATRA2 model transformation framework were overridingly useful for the contributions of this thesis as well.

There were many memorable moments during my visit in Darmstadt. I say many thanks to Janusz Szuba for the common sightseeings and excursions, to Carsten Amelunxen for organizing the beer tasting events, to Jan Schluchtmann for the interesting conversations, and to all the members of the Real-Time Systems Lab for the "table soccer" tournaments and for the kindness preventing me to feel alone.

At this point, I should acknowledge many researchers including at least Gernot Veit Batz (University of Karlsruhe), Marita Breuer (RWTH Aachen University), Rubino Geiß (University of Karlsruhe), Tom Mens (University of Mons-Hainaut), Arend Rensink (University of Twente), Olga Runge (TU Berlin), Christian Schneider (University of Kassel), Adam Szalkowski (University of Karlsruhe), Gabriele Taentzer (Philipps-Universität Marburg), Hans Vangheluwe (McGill University), Kang Zhang (University of Texas at Dallas), Albert Zündorf (University of Kassel) for the fruitful discussions, and/or their encouragement and support.

The current dissertation was partially supported by the SegraVis European Research Training Network, the SENSORIA European IP (IST-3-016004), the Hungarian National Research Fund and the National Office for Research and Technology (Grants No. T030804, T42559, and 67651, OTKA), and the Péter Bizáki Puky Scholarship.

From my private life I would like to say many thanks to Noémi Ambrózy, Egmont Koblinger, Zsófia Müller, and Eszter Sipos Szabó for their love, friendship and continuous support, and to Dániel's family: Szilvia, Balázs and Csaba for the fun moments during the recent years.

Finally, I acknowledge my beloved parents Győző and Mária. Without their love, encouragement and support I would never succeed.

CHAPTER

1 Introduction

1.1 Model-Driven Engineering

Since the design of embedded systems and workflow management necessitate the handling of large system models during the development process, the paradigm of Model-Driven Development (MDD) has recently become a leading trend in software engineering. The aim of MDD is to carry out a thorough system modeling before implementation. Key ideas of MDD are to create models of the software on various abstraction levels and from various viewpoints and to support automatic code generation from these models. The main advantages of the MDD concept are the reuse of high abstraction level models and an increase in productivity by high degree of automation. The idea of MDD is not restricted to software engineering domains, but also applicable e.g., to business modeling [41] and civil engineering [19]. In this sense, MDD fits into the broader concept of Model-Driven Engineering (MDE), which only prescribes the systematic use of models throughout the entire engineering lifecycle.

The best-known realization of both MDD and MDE principles is the Model-Driven Architecture (MDA) [78, 101] initiative of the Object Management Group (OMG). The aim of MDA is to separate software or business functionality from platform details, which is achieved as follows.

The conceptual design of functionalities of the software or business system is captured in the form of a *platform independent model* (PIM), which constitutes a reusable model represented on a high abstraction level. PIMs can survive changes in realization technologies and software architectures.

A *platform-specific model* (PSM) is also a model of the system under design, but in addition, it is linked to an underlying technological platform such as a specific execution platform, software architecture, operating system or database. In this sense, systems described in the CORBA Interface Definition Language [104], business functionalities defined by Enterprise Java Beans [130] interfaces, or even database schemas specified by Oracle specific data definition statements [107] can be considered as platform-specific models.

According to the envisioned engineering process of MDA, the PIM is designed first. Then a PSM is produced from the PIM by an automated model transformation [13]. Finally, program code is generated automatically from the PSM to provide an implementation for the system.

MDA uses the following standards for specifying models of concrete systems and structural definitions of application domains.

- Unified Modeling Language (UML) [139] provides a visual modeling language for the analysis, design, implementation, deployment, and documentation of applications.
- Meta-Object Facility (MOF) [103] specifies repositories for domain-specific applications and modeling languages by constructing structural descriptions called metamodels.
- XML Metadata Interchange (XMI) [105] is a metamodel-specific XML format for interchanging models between different CASE tools;
- Common Warehouse Metamodel (CWM) [100] serves as a language for database integration in data mining and warehousing.

MDA tools [86, 135] should provide support for (i) creating and editing models, (ii) checking completeness and consistency, (iii) calculating metrics, (iv) transforming models to other models or program code, (v) composing several source models, (vi) model-based testing, (vii) simulating the execution of the systems represented by models, and (viii) re-engineering by transforming legacy systems to well-formed models.

In order to carry out all these tasks, a large number of modeling languages and tools are used in a typical model-driven development process. Powerful domain-specific modeling environments (such as the GMF in Eclipse, or the DSM framework of Microsoft Visual Studio) frequently provide rich support for developing editors, and code generators. However, the design of model transformations are usually not supported properly in these industrial tools. This thesis primarily focuses on to provide advanced support for executing complex model transformations within and between these modeling languages.

1.2 Model transformation

Model transformation plays a key role in the overall process of MDA. The aim of model transformation is to carry out automated translations within and between modeling languages. The MDD approach requires these transformations to be

- captured by a high-level, MOF-compliant, declarative specification language,
- automatically executed by efficient algorithms and techniques,
- incremental in nature by propagating modifications in the source model to the target model,
- bidirectional,
- traceable,
- defined by reusable and extendable transformation specifications,
- executed in a transactional context, and
- extensively supported by industrial quality tools.

1.2.1 Query/Views/Transformations – the model transformation standard

Query/Views/Transformations (QVT) [109] is a standard being published recently by the Object Management Group (OMG) for specifying multi-directional model transformations.

QVT consists of several specification languages providing both declarative and imperative ways to define transformations.

Declarative languages. Declarative languages are organized in a two layer hierarchy based on their level of abstraction. The more abstract, user-friendly Relations Language (i) enables the specification

of complex pattern matching tasks, (ii) provides a template-based object creation mechanism, and (iii) creates trace classes implicitly. Furthermore, it defines both a graphical and a textual notation for its concrete syntax. On the other hand, the Core Language is only a minimal extension to the Essential MOF (EMOF) part of MOF 2.0 [103] and Object Constraint Language (OCL) [102], which describe models and constraints on models, respectively. In the Core Language, only patterns with simple structure can be matched.

Imperative languages. Imperative transformations can also be specified in QVT by its Operational Mappings Language, which uses OCL expressions with side effects. This complements the Relations Language by providing an equivalent, imperative description for its declarative relations.

Since QVT is a recent standard with a short history, only initial prototypes (e.g., mediniQVT [73], MTF [2]) have been developed, and efficient implementations scaling up to complex model transformation are still missing.

While the model transformation community lacks QVT-based tools, a large variety of tools using different concepts and techniques have already been implemented. An overview on these tools together with their categorization is now presented.

1.2.2 Model transformation tools

Since [30] provides an excellent and wide range survey on the categorization of existing model transformation tools, this subsection only gives a brief summary focusing on those groups that are related to the topics discussed in the current thesis.

Based on the kind of input and output of the tool, several scenarios are distinguished including model-to-model, model-to-code, code-to-model transformations and inter-model rewriting. Since a well-defined abstract syntax for the program code (together with appropriate parsing methods) can also be considered as a model, only the most general and challenging case, namely tools performing model-to-model transformations are examined in the following.

Based on the categorizaton of [30], model-to-model transformation tools can be further grouped to the following subcategories. (i) Direct manipulation approaches (like Jamda [18]) simply use an internal representation for storing models and some APIs for manipulation. (ii) Relational approaches (such as QVT [109]) specify transformations by stating relations between source and target model elements. Related tools may apply logic programming [53] for implementation purposes. (iii) The declarative and rule-based approach of graph transformation can also be an underlying implementation technique for model transformation. Since the current thesis is built on this last technique, it is discussed in details in Section 1.3. (iv) Hybrid approaches (like ATL [14]) combine different techniques from the above-mentioned ones. In addition to the these categories, the transformation framework of Common Warehouse Metamodel (CWM) specification [100] and translations, which use Extensible Stylesheet Language Transformations (XSLT) [159] for the implementation also belong to the group of model-to-model transformations.

1.3 Graph transformation as a model transformation approach

Graph transformation (GT) [38, 118] provides a declarative language for specifying the manipulation of graph models by means of GT rules, which consist of a left-hand side (LHS) and a right-hand side (RHS) graph. Model manipulation is performed by searching for such parts of the model that can be matched to the LHS in the pattern matching phase, and by modifying these selected parts based on the difference of LHS and RHS in the updating phase.

Due to its declarative and rule-based nature, graph transformation shows similarity to the Relations Language of QVT, but GT additionally supports control structures for guiding the execution of elementary model transformation steps. Its history is dated back to the 1970s well before the MDD paradigm has been evolved as indicated by [39]. Since then, graph transformation has proved its maturity in the specification of visual languages [8, 9, 161] and the prototyping of visual language tools [96], and it has become a popular technique for capturing model transformations as well [30, 40].

1.3.1 Architecture of a model transformation tool

The typical architecture of an exogeneous model transformation tool is presented in Fig. 1.1.

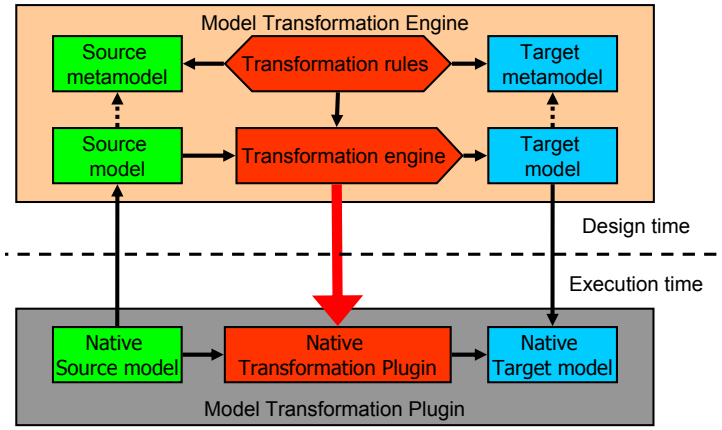


Figure 1.1: The architecture of an exogeneous model transformation tool

At design time, the transformation is specified by means of GT rules, which have left-hand side and right-hand side graphs consisting of nodes and edges from both the source and target modeling domains (i.e., from the metamodels). These rules drive the transformation engine by converting well-formed models of the source domain to models of the target domain. In this case, both models and the transformation engine itself can be considered as parts of the tool.

Since legacy applications with own source and target model representations often need transformation support as well, the architecture should be able to provide generated, native transformation plugins, which can later be integrated into both standard and non-standard environments for execution purposes.

1.3.2 Categorization of graph transformation tools

Graph transformation tools can be categorized according to their execution mode and the underlying pattern matching strategy. For further comparison of graph transformation approaches see [121].

Categorization of execution modes. Based on their execution mode, we distinguish between interpreted and compiled approaches.[1]

- *Interpreted* approaches use an underlying graph transformation interpreter, which gets a compile-time preprocessed, representation of GT rules for determining the activities to be executed at run-time.

[1]Some tools are hybrid in a sense that they combine compilation with interpretation (e.g., PROGRES generates graph machine byte code that is either interpreted or compiled into C or Java code).

- *Compiled* approaches perform graph transformation by programs that are directly executable on the target machine at run-time without the need for an underlying GT interpreter. At compile-time, these approaches generate source code (e.g., C, C++, or Java) that describes the activities to be performed during graph transformation. Then, this source code is compiled by a traditional compiler.

As stated in [7], model transformation is becoming an engineering discipline, and, thus, it requires conceptual and tool support for its entire life-cycle (including the specification, design, execution, validation and maintenance of transformations). Since these tasks set up conflicting requirements, it is difficult to find the best compromise. By using the above categorization, interpreted approaches have a clear advantage during the validation (e.g., by interactive simulation) or the maintenance phase of model transformations due to their flexibility. On the other hand, compiled approaches are advantageous, when performance is the key issue like in case of transformation execution.

Categorization of pattern matching strategies. Pattern matching algorithms also have two groups.

- Algorithms based on *constraint satisfaction* interpret LHS as a set of constraints, which should be satisfied by the matchings provided by these algorithms.
- Algorithms based on *local searches* start from matching a single node and extending the matching step-by-step by mapping neighboring nodes.

1.4 Problem statement

After examining several tools (which are going to be surveyed in Section 3.3) I have discovered the following problems, which have become the challenges for my research.

- **Lack of objective measurements for memory and time.** Only estimates existed about the memory usage and the run-time performance of model transformation tools, and their exact characteristics have never been objectively assessed due to lack of measurements.
- **Insensitivity of algorithms to models under transformation.** Optimization of pattern matching algorithms only exploited restrictions imposed by the problem domain, but ignored any additional information about the model under transformation.
- **Insensitivity of algorithms to transformation flow.** Pattern matching algorithms were inflexible and insensitive to the transformation flow, although significant changes might be experienced in the structure and size of instance models, while the transformation progresses from the beginning to the end. Moreover, they could not either be tuned to efficiently handle the situation, when models remained nearly intact while executing a few subsequent steps of the transformation,
- **Performance problems when transforming large models.** All the examined tools performed in-memory translations, and no analysis investigated the transformation of huge system models, which were unable to fit into the main memory.
- **Integration of transformations to existing tools.** Although a large variety of highly sophisticated standalone graph transformation solutions were available, their integration into existing MDA tools was inhibited by the missing support for transformation APIs.

As a conclusion, support for at least the above-mentioned performance and tool integration related issues was missing from both the design and the implementation of such model transformation algorithms and tools that were aimed to be applied in industrial, software engineering projects.

Thesis objectives

In this thesis, I propose several advanced techniques for the implementation of model transformation systems. More specifically, in addition to the elaboration of algorithmic aspects of the suggested approaches, my aim is to also examine their practical considerations including performance measurements and the analysis of their tool integration capabilities.

My contributions are the following.

- **Contribution 1.** I propose a benchmarking framework to quantitatively assess the run-time performance of graph transformation tools. This benchmarking framework identifies transformation problem-specific and tool-specific characteristics, which have significant impact on the performance of transformations. Additionally, it specifies benchmark examples for a model transformation and a simulation scenario.

- **Contribution 2.** I present a technique for implementing graph transformation built on top of relational database management systems (RDBMS) by performing all the calculations on data stored on disks. In the proposed approach, graph pattern matching is implemented by executing queries, while the model updates are represented by performing data manipulation statements (such as insert, delete, or update).

- **Contribution 3.** I establish model sensitivity by employing statistics being collected from concrete typical models of the domain for providing better cost functions for optimization. Additionally, I propose an adaptive approach, where the optimal strategy can be selected from precompiled methods at run-time based on statistics of the model under transformation. These suggestions enable the adaptive behaviour of pattern matching algorithms making them sensitive to run-time models and to the transformation flow.

- **Contribution 4.** I propose an incremental approach, which stores the partial matchings of earlier graph transformation steps in memory and updates these data structures in an incremental way in response to model modification triggers. This solution significantly accelerates the graph pattern matching phase for the price of increased memory usage, which is especially suitable for model transformations, which only perform small manipulations in subsequent steps.

I demonstrate all the concepts and contributions of this thesis on well-known model transformation problems like the object-relational mapping of [49], and the distributed mutual exclusion algorithm of [60], which are going to be specified later in Sections 2.2 and 3.2, respectively.

1.5 The structure of the thesis

The current thesis is structured into nine main chapters (including this introduction) that contain overviews and new results and two appendices complementing the main parts with additional information.

- Chapter 2 gives an introduction to the basics of modeling language specification.

- Chapter 3 presents the technique of graph transformation.
- Chapter 4 introduces the basic principles of implementing graph transformation and discusses considerations about the efficiency of pattern matching algorithms.
- Chapter 5 presents a benchmarking framework for model transformation tools (as suggested by Contribution 1) to quantitatively assess and analyze their run-time performance and to compare different optimization strategies used for of pattern matching.
- Chapter 6 proposes a novel approach for graph transformation built on top of standard relational database management systems, which corresponds to Contribution 2.
- Chapter 7 presents a technique to implement adaptive and model-sensitive graph pattern matching modules (as suggested by Contribution 3), which use the statistics of the instance model under transformation to dynamically select the optimal from precompiled strategies.
- Chapter 8 presents the foundations of an incremental graph pattern matching engine (as proposed by Contribution 4), which keeps track of existing matchings in an incremental way to reduce the execution time of graph pattern matching.
- Finally, Chapter 9 concludes the main parts of the current thesis.

Notational guide

In order to obtain a consistence appearance of the thesis, the following rules are followed.

- This thesis is mainly written in third person singular. In conclusions after each chapter, I emphasize my own contribution by first person singular or plural.
- **Terms in formal definitions** are printed in **bold** letters.
- New *concepts*, *informal definitions* and *theorems* are typeset in *italics*.
- `Code extracts` always appear as `typewritten` text in listings with grey background.
- For referring to texts in figures, sans serif fonts are used.

CHAPTER

2 Graph Models

In this chapter, the basics of modeling language specification are introduced. Concepts are presented on the object-relational mapping [49], which is reused as a running example in later chapters. Finally, in order to demonstrate the practical links of the theoretical concepts, a corresponding Java based implementation is presented.

2.1 Metamodels

This section summarizes the foundations of modeling language specification. The abstract syntax of a modeling language (or domain) is described by the *metamodel*. Nodes of the metamodel are called *classes*. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some additional ones. *Associations* define connections between classes. Both ends of an association may have a *multiplicity constraint* attached to them, which declares the number of objects that, at run-time, may participate in an association. We consider the most typical multiplicity constraints, which are the at most one (denoted implicitly by diamonds for containment, or explicitly by the 0..1 notation for general associations at the corresponding line end), and the arbitrary (denoted uniformly by line ends with * multiplicity).

While attribute handling is an important practical aspect, we decided not to include it in the formalization of metamodels presented in the current chapter. Since the core ideas of the current dissertation can be discussed without this issue, we believe that this simplification in the formal treatment does not have significant impact. Note that attribute handling was, in fact, implemented in the relational database approach (Chapter 6), and it was formally discussed in [151].

Example 1 For presenting metamodeling concepts, the object-relational mapping [49] has been selected as a running example. In this domain, UML class diagrams are transformed to relational data models, according to the following guidelines. Packages and classes of the class diagram are converted one by one to database schemas and tables, respectively. Each association is transformed to a corresponding table as well. Each table has a column with primary key for storing identifiers, and one column for each attribute of the original class or association. Inheritance is reflected in relational databases as foreign key constraints. Finally, structural well-formedness criteria defined by association ends are also represented by foreign key constraints in the database.

The metamodels of UML class diagrams and relational database schemas (following the CWM standard [108]) are depicted in Fig. 2.1. In order to avoid complex figures, only the relevant parts of the metamodel are presented.

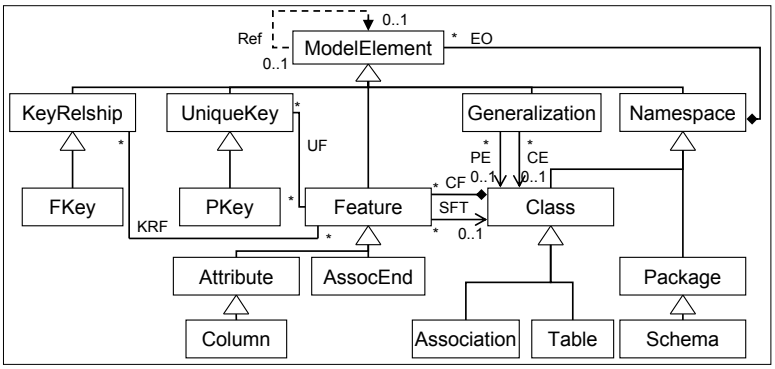


Figure 2.1: Metamodel of the problem domain

The metamodel consists of 16 classes including elements of UML class diagrams such as Feature, Class, and Association, and of relational database concepts such as Table, Schema, and Column. Note that the CWM standard derives database notions from UML notions by inheritance as in case of class Column, which is inherited from class Attribute (see Fig. 2.1).

Association (structural feature type) SFT connects classes Feature and Class. An at most one multiplicity constraint can be found at the end of association SFT that is next to class Class expressing that each Feature can have at most one Class as its type. However, in the other direction, an arbitrary multiplicity constraint holds, which means that each Class may denote the type of many Features.

Furthermore, we use reference edges Ref (denoted by dashed lines in instance models) for connecting source and target model nodes despite they are not included in the CWM standard.

Formalization

Now the formalization of concepts related to metamodels is presented by following the notation of [151].

Definition 1 A **directed graph** (denoted by $G = (V_G, E_G, src_G, trg_G)$) is a 4-tuple, where V_G and E_G denote nodes and edges of the graph, respectively. Functions $src_G : E_G \rightarrow V_G$ and $trg_G : E_G \rightarrow V_G$ map edges to their source and target node, respectively.

Definition 2 A **metamodel** (denoted by MM) is a directed graph, where

- V_{MM} and E_{MM} denote nodes and edges of the metamodel;
- a **class** C is a node of the metamodel, formally, $C \in V_{MM}$;
- associations ($Assoc$) and generalization (inheritance) edges ($Inher$) constitute a (distinct and complete) partition of edges, formally, $E_{MM} = Assoc \cup Inher, Assoc \cap Inher = \emptyset$;
- associations can be further partitioned into 'many-to-many' ($Assoc_{M2M}$) and 'many-to-one' ($Assoc_{M2O}$) associations, formally, $Assoc = Assoc_{M2M} \cup Assoc_{M2O}, Assoc_{M2M} \cap Assoc_{M2O} = \emptyset$

  - a **many-to-many association** A **from source class** C_s **to target class** C_t (denoted by $C_s \xrightarrow{A}_* C_t$) is an edge from the set $Assoc_{M2M}$, where $src_{MM}(A) = C_s \in Cls$, $trg_{MM}(A) = C_t \in Cls$;
  - a **many-to-one association** A **from source class** C_s **to target class** C_t (denoted by $C_s \xrightarrow{A}_1 C_t$) is an edge from the set $Assoc_{M2O}$, where $src_{MM}(A) = C_s \in Cls$, $trg_{MM}(A) = C_t \in Cls$;

- a **generalization (inheritance) edge** I **leading from class** C_t **to class** C_s (denoted as in UML by $C_s \leftarrow C_t$) is an edge of set $Inher$, formally, $src_{MM}(I) = C_t \in Cls$, $trg_{MM}(I) = C_s \in Cls$, and $C_s \leftarrow C_t \in Inher$.

In the above definition, associations define binary relations between classes. In this thesis, we do not handle association classes. In the following, the notation $C_s \xrightarrow{A} C_t$ is used for a general association of any kind that is $A \in (Assoc_{M2M} \cup Assoc_{M2O})$.

Inheritance graph. The inheritance hierarchy forms a lattice, which implies that the inheritance graph is a directed acyclic graph (DAG), and there is a common root ancestor class for all classes.

Definition 3 The **inheritance graph** $MM_{Inher} = (Cls, Inher, src_{MM}, trg_{MM})$ is the type graph restricted to generalization (inheritance) edges, which forms a lattice.

Definition 4 Given a metamodel MM, class C_1 is a (direct) **superclass** of class C_2 (or, equivalently, class C_2 is a (direct) **subclass** of class C_1) as denoted by $C_1 \leftarrow C_2$, if and only if

- there is a generalization edge $C_1 \leftarrow C_2 \in Inher$;
- there are no other classes in the inheritance hierarchy between C_1 and C_2, formally, $\nexists C \in V_{MM}$ such that $C_1 \leftarrow C \leftarrow C_2$.

Note that this definition does not imply that a class C_2 has a single superclass C_1, as multiple inheritance is allowed in the inheritance graph. Since the superclass of a class may also have its own superclass, it is useful to define the transitive closure of the superclass relation.

Definition 5 Given a metamodel MM, class C_1 is an **ancestor (class)** of class C_2 (or, equivalently, class C_2 is a **descendant** of class C_1) (denoted by $C_1 \overset{*}{\leftarrow} C_2$), if either $C_1 = C_2$, or $\exists C \in V_{MM}$ such that $C_1 \leftarrow C \overset{*}{\leftarrow} C_2$.

Capital letters from the beginning of the alphabet (e.g., $C, D_s \xrightarrow{A} D_t$) will be used for meta-level graph elements (classes, associations).

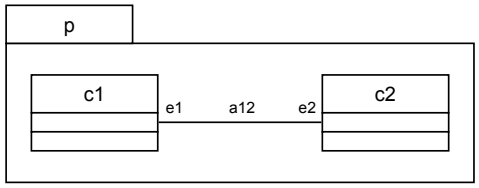
2.2 Instance models

The *instance model* is a graph that describes concrete systems defined in a modeling language. Its nodes and edges are called *objects* and *links*, respectively. The instance model is a *well-formed* instance of the metamodel, which means the fulfillment of the following criteria.

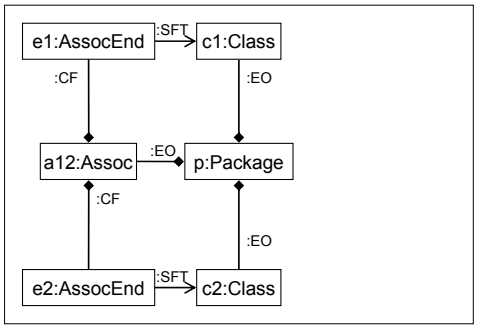
- Objects and links are instances of metamodel level classes and associations, respectively.
- Source and target object of each link have to conform to source and target class of the association from which the link has been instantiated.

- Each object may have only one such outgoing link that has been instantiated from an association with a many-to-one multiplicity constraint at its target end.
- No parallel links of the same type are allowed between any pair of objects.

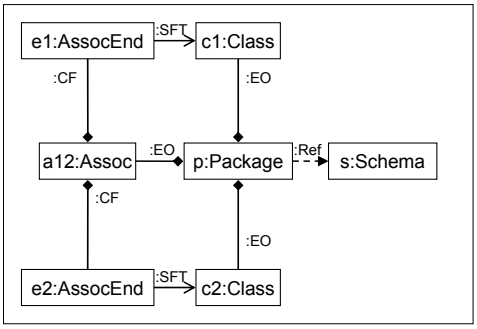
Example 2 A well-formed instance model of this domain is presented by the concrete syntax of UML class diagrams in Fig. 2.2(a) and by a corresponding abstract syntax representation in Fig. 2.2(b). The instance model has a UML[1] package p, which contains UML classes c1 and c2, and a UML association a12. Containments are expressed by the element ownership edges EO. UML association a12 connects UML classes c1 and c2. This connection is denoted by UML association ends e1 and e2, which are contained by UML association a12 as classifier features CF, and which mark UML classes c1 and c2 as their structural feature types SFT, respectively.



(a) UML class diagram (concrete syntax) representation of Model1



(b) Model1



(c) Model2

Figure 2.2: Sample instance models

Formalization

Now the formalization of concepts related to instance models is presented.

Definition 6 The **model element universe** (denoted by $\mathfrak{U}$) is an infinite set that contains all the valid identifiers that can appear in a model. The set of **unused model element universe** (denoted by $\mathfrak{U}_{NU}$) is such an infinite subset of the model element universe that contains those identifiers that have not yet appeared in the model. Formally, $\mathfrak{U}_{NU} \subseteq \mathfrak{U}$.

Definition 7 Given a metamodel MM, a **well-formed instance model (graph)** M **of the metamodel** MM is a directed graph together with a **direct type function (graph morphism)** $t : M \rightarrow MM$, which maps model M to metamodel MM according to the following rules

[1]To prevent confusion between metamodeling terms and class diagram notions we use the UML prefix for the latter.

- **Unambiguous mapping of objects:** model nodes (called as **objects**) are mapped to metamodel nodes, formally, $\forall c \in V_M : t(c) \in V_{MM}$;
- **Unambiguous mapping of links:** model edges (called as **links**) are mapped to associations, formally, $\forall e \in E_M : t(e) \in Assoc$;
- **Usage of identifiers for objects and links:** objects and links are always picked from the model element universe, but they never belong to its unused counterpart, formally, $V_M \subseteq \mathfrak{U} \setminus \mathfrak{U}_{NU}$, and $E_M \subseteq \mathfrak{U} \setminus \mathfrak{U}_{NU}$;
- **Type conformance of source objects:** the direct type of the source object of a link is a descendant of the source of the direct type of the same link, formally, $\forall e \in E_M : src_{MM}(t(e)) \overset{*}{\leftarrow} t(src_M(e))$;
- **Type conformance of target objects:** the direct type of the target object of a link is a descendant of the target of the direct type of the same link, formally, $\forall e \in E_M : trg_{MM}(t(e)) \overset{*}{\leftarrow} t(trg_M(e))$;
- **Multiplicity criterion for many-to-one associations:** each object can have at most one link of a given direct type originating from the same many-to-one association. Formally, $\forall A \in Assoc_{M2O}, \forall e_1, e_2 \in E_M : src_M(e_1) = a \wedge trg_M(e_1) = b \wedge src_M(e_2) = a \wedge trg_M(e_2) = c \wedge A = t(e_1) = t(e_2) \Longrightarrow e_1 = e_2$; and
- **Non-existence of parallel edges of same type:** No parallel edges are allowed, which means that there cannot be any pair of links of the same type leading between the same pair of objects in a given direction. Formally, $\forall e_1, e_2 \in E_M : src_M(e_1) = src_M(e_2) \wedge trg_M(e_1) = trg_M(e_2) \wedge t(e_1) = t(e_2) \Longrightarrow e_1 = e_2$.

Small letters from the beginning of the alphabet (e.g., $c, a \xrightarrow{e} b$) will be used for objects and links of the instance model.

In the following, we use terms **many-to-many link** (denoted by $a \xrightarrow{e}_* b$), and **many-to-one link** (denoted by $a \xrightarrow{e}_1 b$), if the direct type of the given link is a many-to-many association, and a many-to-one association, respectively.

Type definition can be generalized in such way that all ancestors of a direct type are also implied.

Definition 8 Given a metamodel MM, a well-formed instance model M with a direct type function t, the **type of an object** c (denoted by $t^*(c)$) consists of all ancestors of $t(c)$. Formally, $t^*(c) = \left\{ C \mid C \in V_{MM} \wedge C \overset{*}{\leftarrow} t(c) \right\}$.

Although our presented graph-based model lacks several advanced features (like the three layer node inheritance of graph schemas [75], or the edge inheritance of type graphs [134]), it is still suitable for transforming models used in practice as demonstrated by the running examples.

2.3 Metamodel and model representation in Java

Java 2 Standard Edition 5.0 (J2SE) [128] and Java 2 Enterprise Edition 5.0 (J2EE) [127] are used as underlying implementation platform in order to demonstrate the strong practical links of the theoretical concepts and algorithms presented in this thesis. Java 2 has been selected due to its "write once run everywhere" philosophy, which makes Java programs portable by allowing their execution on a variety of hardware platforms and operating systems.

2.3.1 Java 2 Platform 5.0 Standard and Enterprise Editions

Java 2 Standard Edition enables the development and deployment of Java applications on desktops and servers as well as embedded and real-time environments. It includes the Java Database Connectivity (JDBC) API, which provides universal data access in Java from a large variety of sources including relational databases, spreadsheets and flat files. In this sense, J2SE provides appropriate support, when models are stored and manipulated in main memory, or when models are stored in an underlying relational database, but they are directly accessed and manipulated by low-level SQL commands.

Java 2 Enterprise Edition defines a layered architecture for scalable, distributed application development including several Java standards and APIs. An enterprise application being developed on the J2EE platform consists of Enterprise Java Beans (EJBs) as its most fundamental building blocks representing business data and functionality. An enterprise application is deployed to and executed by an application server, which provides many high-level services (such as transactions, security, persistence, etc.) beyond the execution of applications.

The following three types of EJBs have been defined by J2EE.

- *Entity beans* are persistent objects representing business data, which are kept synchronized with an underlying relational database by means of an object-relational mapping. Entity beans are uniquely identified by their primary key and they can be in relationship with other entity beans referring to each other by direct references (many-to-one or one-to-one relationships) or typed collections (many-to-many or one-to-many relationships).

- *Session beans* implement the business functionality of the application. They can be considered as simple collections of business methods. Depending on whether they should preserve any information between consecutive invocations of their methods, stateful and stateless session beans are distinguished. As our approach does not require any transformation related information to be stored, we use stateless session beans.

- *Message driven beans* are EJBs that provide asynchronous message processing functionality. They can be considered as listeners to which application clients and other EJBs can send messages.

J2EE is used as an underlying platform, when models are stored in a relational database and they are accessed and manipulated as plain old Java objects through transparent object-relational mapping and persistence layers.

2.3.2 Mapping metamodels to EJB3 entity bean classes

Now I present a method for mapping metamodels to EJB3 entity bean classes. Although the concrete example produces EJB3 entity beans as output, the generated interfaces show high similarity to both (plain old) Java objects (POJOs) with property accessor methods and interfaces specified by relevant modeling frameworks (such as Eclipse Modeling Framework (EMF) [136] and Java Metadata Interface (JMI) [129]). As a result, the presented techniques could easily be adapted to one framework or the other.

Based on the metamodel, entity bean classes are generated by using the standard object-relational mapping of [130], which can be summarized as follows. A class of the metamodel is mapped to an entity bean class. The inheritance relations between classes are maintained accordingly. Each association end with an at most one (arbitrary) multiplicity constraint is mapped to a Java attribute (collection) and two corresponding property accessor (i.e., a getter and a setter) methods in the entity

bean class that represents the metamodel class being located at the opposite end of the association. A Java attribute `id` representing the unique identifier and its two corresponding property accessor methods are added to each entity bean class that does not have a superclass.

In order to specify how Enterprise Java Beans are deployed to the application server, we use Java annotations for marking up class declarations, fields, methods, and other program elements in the source code. In this sense, annotations `@Entity` or `@Stateless` at class declarations denote that the given class should be handled as an entity bean or a stateless session bean, respectively. Annotation `@Inheritance` influences how inherited classes are mapped in the underlying relational database. Annotations `@ManyToMany`, `@ManyToOne`, `@OneToMany`, `@OneToOne` mark up getter methods defined for each association end. to specify multiplicity criteria of the corresponding association.

Example 3 The skeleton of the entity bean class representing a Feature is shown by Listing 2.1.

```
@Entity                                                 // Feature is an entity bean.
@Inheritance(strategy = InheritanceType.JOINED) // A different table is used
                                                        // for each class
                                                        // in the inheritance hierarchy.
public class Feature extends ModelElement {
   // Attributes
   private Class cf;
   private Class sft;
   private Collection<UniqueKey> uf = new ArrayList<UniqueKey>();
   private Collection<KeyRelationship> krf = new ArrayList<KeyRelationship>();

   // Property accessor methods
   @ManyToOne // Association CF has * multiplicity constraint on its Feature side,
              //                  and 1 multiplicity constraint on its Class side.
   public Class getCF() { return cf; }
   public void setCF(Class cf) { this.cf = cf; }

   @ManyToOne
   public Class getSFT() { return sft; }
   public void setSFT(Class sft) { this.sft = sft; }

   @ManyToMany(mappedBy="uf") // Association UF has * multiplicity constraint
                              // on both its Feature and UniqueKey sides, and
                              // class on the other side (i.e., UniqueKey)
                              // is responsible for handling links of type UF.
   public Collection<UniqueKey> getUF() { return uf; }
   public void setUF(Collection<UniqueKey> uf) { this.uf = uf; }

   @ManyToMany(mappedBy="krf")
   public Collection<KeyRelationship> getKRF() { return krf; }
   public void setKRF(Collection<KeyRelationship> krf) { this.krf = krf; }
}
```

Listing 2.1: Skeleton of the Feature entity bean class

As Feature is a subclass of ModelElement, the identifier attribute `id` has not been created. According to the metamodel of Fig. 2.1, the Feature class has four incident edges. Consequently, the generated code has four attributes and eight accessor methods.

In order to get an in-memory model space representation, which contains plain old Java objects, only annotations have to be removed from the above specification.

2.3.3 Creating sample models as EJB3 entity bean instances

Instance models representing the system under design are stored in an underlying database of the application server. By using entity beans, objects of the instance model can be created, accessed and manipulated exactly the same way as traditional (plain old) Java objects with the single exception that these objects have to be explicitly persisted by calling the `persist()` method of the entity manager, which in turn, provides the persistence context for the entity beans.

Example 4 The Java code that produces Model 1 of Fig. 2.2(b) in an application server is presented by Listing 2.2.

```
1  @PersistenceContext
2  EntityManager em;
3  // Creating new objects
4  Package p = new Package();
5  Class c1 = new Class();
6  Class c2 = new Class();
7  Association a12 = new Association();
8  AssocEnd e1 = new AssocEnd();
9  AssocEnd e2 = new AssocEnd();
10 // Creating new links
11 c1.setEO(p);
12 c2.setEO(p);
13 a12.setEO(p);
14 e1.setCF(a12);
15 e1.setSFT(c1);
16 e2.setCF(a12);
17 e2.setCF(c2);
18 // Persisting new objects
19 entityManager.persist(p);   entityManager.persist(a12);
20 entityManager.persist(c1);  entityManager.persist(c2);
21 entityManager.persist(e1);  entityManager.persist(e2);
```

Listing 2.2: Java code for generating Model 1 of Fig.2.2(b)

In Lines 1–2, the entity manager is initialized by the application server by injecting a persistence context (by using the annotation `@PersistenceContext`) for the method that creates entity beans. Then 6 objects and 6 links are created by Lines 3–9 and Lines 10–17, respectively. Finally, the new objects are persisted by the underlying database of the application server while executing Lines 18–21.

2.4 Conclusion

In this chapter, modeling language specification has been surveyed by introducing metamodels for describing modeling domains, and instance models for specifying concrete systems. These concepts have been presented by using the object-relational mapping as a running example. Finally, metamodels and models have been exemplified in an EJB3 environment to illustrate their practical applicability by using entity bean classes and instances.

CHAPTER

3 Computing by Graph Transformation

In this chapter, it is demonstrated how model transformations between modeling languages can be specified by means of graph transformation. Concepts are again presented on the object-relational mapping [49]. Additionally, a distributed mutual exclusion algorithm [60] is introduced as a second running example to represent a different application scenario of graph transformation. Then graph transformation tools are surveyed and some representatives are selected for being used in performance measurement and analysis. Finally, the architecture of a modular graph transformation engine is presented.

3.1 Graph transformation

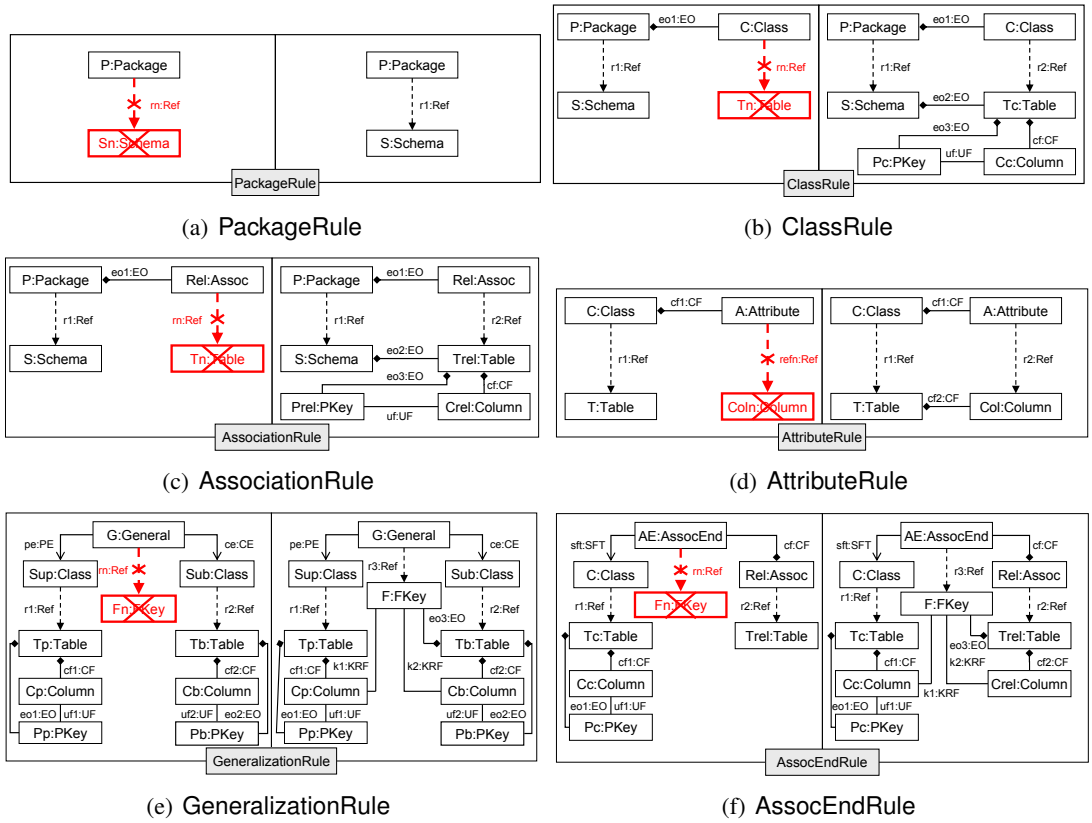
We present how model transformation between modeling languages can be specified by graph transformation [38, 118]. In this dissertation, the single-pushout approach is used with injective rules and matchings. Negative application conditions and additional constraints like multiplicities and the non-existence of parallel edges of the same type are also considered. As already mentioned in Sec. 2.1, attribute handling is not discussed in the formal treatment, since the core ideas of the current dissertation can be presented without dealing with this issue.

3.1.1 Graph transformation rules

A *graph transformation rule* contains a *left–hand side graph* LHS, a *right–hand side graph* RHS, and *negative application condition graphs* NAC [58] (depicted by crosses). The LHS and the NAC graphs are together called the *precondition of the rule*. In the following, terms LHS, RHS, and NAC *patterns* are used interchangeably with LHS, RHS, and NAC graphs, respectively. These new terms are introduced only to avoid the excessive use of the word "graph" in the upcoming chapters and to ease the identification of concepts that are part of graph transformation rules.

Example 5 The object-relational mapping can be described by 6 graph transformation rules as presented in Fig. 3.1.

(a) PackageRule (Fig. 3.1(a)) simply generates a database schema for a UML package.



(a) PackageRule (b) ClassRule

(c) AssociationRule (d) AttributeRule

(e) GeneralizationRule (f) AssocEndRule

Figure 3.1: Rules describing the object relational mapping

(b) ClassRule (Fig. 3.1(b)) searches for a UML class in the UML package, for which there does not exist a corresponding table in the database schema, and creates the corresponding table that has a single column Cc, for which a primary key Pc is defined.

(c) AssociationRule (Fig. 3.1(c)) creates a new table in the database, if there has not been any table assigned yet. This new table has again a single column Crel with a primary key Prel.

(d) A new column is created in the table assigned to the UML class that includes the unhandled UML attribute. This is performed by the AttributeRule (Fig. 3.1(d)).

(e) The inheritance relation in the UML model is handled by appropriate foreign key constraints in the database schema. This is expressed by the GeneralizationRule (Fig. 3.1(e)), which creates a foreign key constraint on the identifier column Cb of the subclass table Tb for any unhandled generalization node. The constraint will refer to the column Cp of the superclass table Tp that has a primary key Pp.

(f) The AssocEndRule (Fig. 3.1(f)) selects an unhandled UML association end, and generates an additional column Crel in the table Trel that has been created for the UML association itself. Moreover, a foreign key constraint is added to the Trel table, which refers to the column Cc of the table Tc that is associated with the UML class C.

As it can be seen in Fig. 3.1, source and target nodes of reference edges represent concepts with UML class diagram and relational database origin, respectively.

Formalization of graph transformation rules

Definition 9 Given a metamodel MM, a **basic rule** r_b consists of a left–hand side graph LHS and a right–hand side graph RHS and an injective partial morphism p : LHS $\rightarrow$ RHS where LHS and RHS are well-formed instances of the metamodel MM.

Definition 10 Given a metamodel MM and a basic rule r_b, a **negative application condition of basic rule** r_b (denoted by $(\mathsf{NAC}, p_{\mathsf{NAC}})$) consists of a negative application condition graph NAC (depicted by crosses in figures), and an injective partial morphism p_{NAC} : LHS $\rightarrow$ NAC where the NAC graph is a well-formed instance of the metamodel MM.

Definition 11 Given a metamodel MM, a **graph transformation rule** $r = (r_b, \{ (\mathsf{NAC}_i, p_{\mathsf{NAC}_i}) \})$ consists of a basic rule r_b, and a set of its negative application conditions $\{ (\mathsf{NAC}_i, p_{\mathsf{NAC}_i}) \}$.

Definition 12 Given a metamodel MM and a graph transformation rule r, the **precondition of rule** r (denoted by r_{PRE}) is the LHS graph together with the set of its negative application conditions.

The LHS graph and the ith negative application condition graph NAC_i of a rule r are denoted by r_{LHS} and r_{NAC_i}, respectively. For the nodes and edges of rules we always use small letters from the end of the alphabet (e.g., $x, u \xrightarrow{z} v$).

3.1.2 A merged graphical representation for rule preconditions

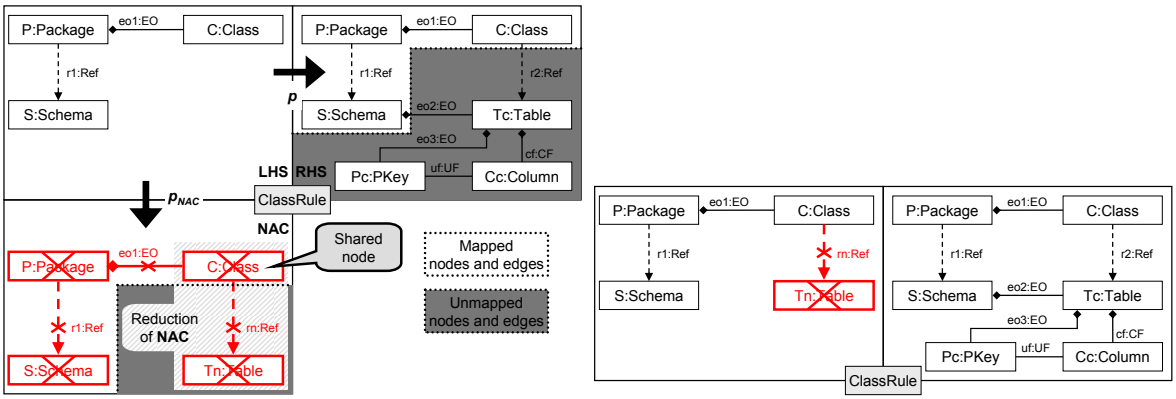
As many of the above graph transformation related concepts are defined by two graphs and a corresponding morphism between them, we introduce a categorization for the nodes and edges of both graphs depending on whether they participate in the morphism (*mapped nodes* and *mapped edges*) or not (*unmapped nodes* and *unmapped edges*).

In order to minimize the overlapping of LHS and NAC patterns (and to later avoid multiple checks caused by this overlapping), we suppose that each original NAC pattern is defined by its *reduction*, which is such a subgraph of the original NAC pattern, which has the minimum number of mapped nodes, and no mapped edges according to the partial morphism, which maps the LHS graph to the NAC graph. As only reduced NAC patterns are used in the rest of this thesis, the word *reduced* is omitted from now on.

In order to have a compact graphical notation for rule preconditions in the figures of this thesis, NAC patterns are graphically represented as if they were merged into the LHS pattern along their mapped nodes. These mapped nodes of the NAC graphs are referred as *shared nodes* due to their visual appearance.

Example 6 The relationship between the traditional and the corresponding merged precondition representation is illustrated on ClassRule in Fig. 3.2. Note that except for Fig. 3.2(a), only the new notation is going to be used in this thesis.

UML package P, UML class C and schema S can be considered mapped nodes according to both morphisms p and p_{NAC} in all the three patterns. Similarly, edge eo1 of type EO and reference edge r1 are mapped edges in the LHS, the RHS, and the NAC graph as well. Table Tn in the NAC pattern is an unmapped node, as it has no origin in the LHS pattern according to morphism p_{NAC}.



(a) Traditional representation of ClassRule

(b) Representation of ClassRule with merged precondition

Figure 3.2: The traditional and the merged visual precondition representation for ClassRule

Although the NAC pattern has four nodes and three edges, its reduction contains only UML class C, table Tn, and the reference edge rn between them. UML class C is a shared node, as it is a mapped node due to having an origin in the LHS graph according to morphism p_{NAC}, and it has an outgoing unmapped edge rn. The precondition of ClassRule is going to be graphically represented in the following in the form of Fig. 3.2(b), which looks as if the reduced NAC pattern of Fig. 3.2(a) was merged with the LHS graph along the shared UML class C.

Formalization of node and edge categorization

Definition 13 Given two directed graphs G_1, G_2, and a partial morphism $p : G_1 \to G_2$, nodes and edges of graphs G_1 and G_2 are classified as follows.

- **Nodes of graph G_1 unmapped by morphism** p (denoted by $V_{G_1} \setminus V_{G_2}$) are such nodes of graph G_1 that *do not have* any image nodes in graph G_2 according to morphism p, formally,

$$V_{G_1} \setminus V_{G_2} = \left\{ x \in V_{G_1} \mid \nexists x' \in V_{G_2} \wedge p(x) = x' \right\}.$$

- **Nodes of graph G_1 mapped by morphism** p (denoted by $\underline{V_{G_1}} \cap V_{G_2}$) are such nodes of graph G_1 that *have* a corresponding image node in graph G_2 according to morphism p, formally,

$$\underline{V_{G_1}} \cap V_{G_2} = \left\{ x \in V_{G_1} \mid \exists x' \in V_{G_2} \wedge p(x) = x' \right\}.$$

- **Edges of graph G_1 unmapped by morphism** p (denoted by $E_{G_1} \setminus E_{G_2}$) are such edges of graph G_1 that *do not have* any image edges in graph G_2 according to morphism p, formally,

$$E_{G_1} \setminus E_{G_2} = \left\{ u \xrightarrow{z} v \in E_{G_1} \mid \nexists u' \xrightarrow{z'} v' \in E_{G_2} \wedge p(u \xrightarrow{z} v) = u' \xrightarrow{z'} v' \right\}.$$

- **Edges of graph G_1 mapped by morphism** p (denoted by $\underline{E_{G_1}} \cap E_{G_2}$) are such edges of graph G_1 that *have* a corresponding image edge in graph G_2 according to morphism p, formally,

$$\underline{E_{G_1}} \cap E_{G_2} = \left\{ u \xrightarrow{z} v \in E_{G_1} \mid \exists u' \xrightarrow{z'} v' \in E_{G_2} \wedge p(u \xrightarrow{z} v) = u' \xrightarrow{z'} v' \right\}.$$

- **Nodes of graph G_2 unmapped by morphism** p (denoted by $V_{G_2} \setminus V_{G_1}$) are such nodes of graph G_2 that *do not have* any origin nodes in graph G_1 according to morphism p, formally,

$$V_{G_2} \setminus V_{G_1} = \left\{ x' \in V_{G_2} \mid \nexists x \in V_{G_1} \wedge p(x) = x' \right\}.$$

- **Nodes of graph G_2 mapped by morphism** p (denoted by $\underline{V_{G_2}} \cap V_{G_1}$) are such nodes of graph G_2 that *have* a corresponding origin node in graph G_1 according to morphism p, formally,

$$\underline{V_{G_2}} \cap V_{G_1} = \left\{ x' \in V_{G_2} \mid \exists x \in V_{G_1} \wedge p(x) = x' \right\}.$$

- **Edges of graph G_2 unmapped by morphism** p (denoted by $E_{G_2} \setminus E_{G_1}$) are such edges of graph G_2 that *do not have* any origin edges in graph G_1 according to morphism p, formally,

$$E_{G_2} \setminus E_{G_1} = \left\{ u' \xrightarrow{z'} v' \in E_{G_2} \mid \nexists u \xrightarrow{z} v \in E_{G_1} \wedge p(u \xrightarrow{z} v) = u' \xrightarrow{z'} v' \right\}.$$

- **Edges of graph G_2 mapped by morphism** p (denoted by $\underline{E_{G_2}} \cap E_{G_1}$) are such edges of graph G_2 that *have* a corresponding origin edge in graph G_1 according to morphism p, formally,

$$\underline{E_{G_2}} \cap E_{G_1} = \left\{ u' \xrightarrow{z'} v' \in E_{G_2} \mid \exists u \xrightarrow{z} v \in E_{G_1} \wedge p(u \xrightarrow{z} v) = u' \xrightarrow{z'} v' \right\}.$$

The above node and edge categories are summarized in Table 3.1.

	In graph G_1		In graph G_2	
	unmapped	mapped	unmapped	mapped
nodes	$V_{G_1} \setminus V_{G_2}$	$\underline{V_{G_1}} \cap V_{G_2}$	$V_{G_2} \setminus V_{G_1}$	$\underline{V_{G_2}} \cap V_{G_1}$
edges	$E_{G_1} \setminus E_{G_2}$	$\underline{E_{G_1}} \cap E_{G_2}$	$E_{G_2} \setminus E_{G_1}$	$\underline{E_{G_2}} \cap E_{G_1}$

Table 3.1: Notational guide summary for the node and edge categories

Definition 14 Given directed graphs G_1 and G_2, and a partial morphism $p : G_1 \to G_2$, **shared nodes of graph** G_2 (denoted by $V_{G_2}^{sh}$) are such mapped nodes of graph G_2, which have at least one incident (incoming or outgoing) unmapped edge in graph G_2. Formally,

$$V_{G_2}^{sh} = \left\{ x \in \underline{V_{G_2}} \cap V_{G_1} \mid \exists y_1 \xrightarrow{z_{in}} x \in E_{G_2} \setminus E_{G_1} \right\} \cup \left\{ x \in \underline{V_{G_2}} \cap V_{G_1} \mid \exists x \xrightarrow{z_{out}} y_2 \in E_{G_2} \setminus E_{G_1} \right\}.$$

Shared nodes are referred as boundary nodes in the double pushout approach [35].

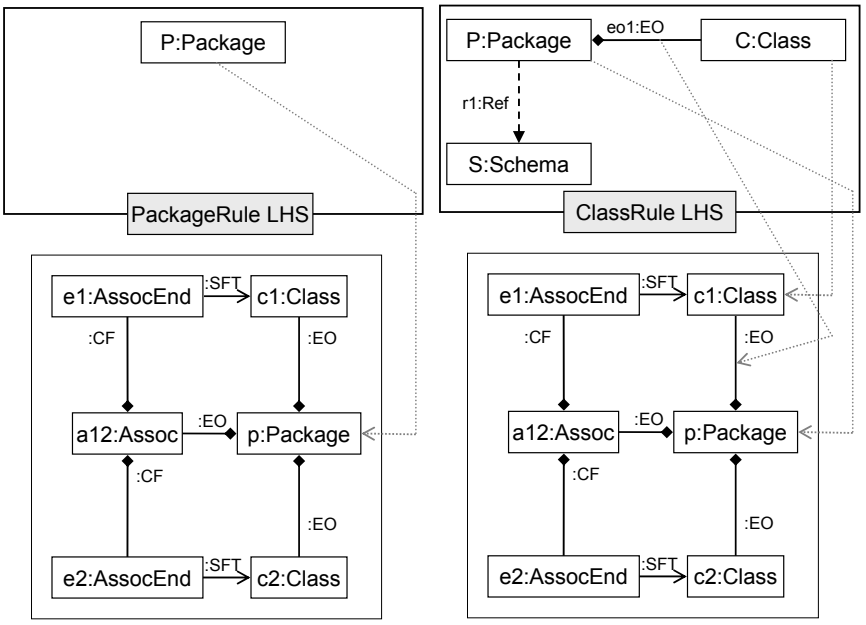
Definition 15 Given two directed graphs G_1 and G_2, and a partial morphism $p : G_1 \to G_2$, the **reduction of graph** G_2 (denoted by G_2^r) is the subgraph of G_2 induced by the shared nodes and the unmapped nodes of graph G_2. Formally, $V_{G_2^r} = V_{G_2}^{sh} \cup (V_{G_2} \setminus V_{G_1})$, and

$$E_{G_2^r} = \left\{ u \xrightarrow{z} v \in E_{G_2} \setminus E_{G_1} \mid u \in V_{G_2^r} \wedge v \in V_{G_2^r} \right\}.$$

3.1.3 Matchings

A *matching* is an injective total graph morphism from a pattern graph to a model, which maps nodes in a type conformant way, and which is source, target, and type preserving in case of edges. A *partial matching for a pattern graph* is a matching for one of its subgraph. A *maximal partial matching* is a non-extensible partial matching. A *matching for a rule in a model* is a matching for the LHS in the model, provided that no matching exists for any embedded NAC graphs.

Example 7 The subgraph of the LHS pattern of ClassRule (Fig. 3.1(b)), which consists of pattern nodes C and P, and the connecting EO edge can be matched to the subgraph of Model 1 (Fig. 2.2(b)) consisting of UML class c1, UML package p, and the connecting EO link, respectively, as shown by the dotted grey edges in Fig. 3.3(b). This set of mappings is a partial matching for the (complete) LHS pattern. It is also a maximal partial matching as it cannot be further extended due to the fact that pattern node S cannot be mapped to any schemas in Model 1. Due to similar reasons, the above set of mappings is not a matching for the (complete) LHS pattern.



(a) Matching for the LHS of PackageRule in Model 1

(b) Matching for the LHS of ClassRule in Model 1

Figure 3.3: Matchings for patterns in Model 1

On the other hand, if UML package p in Model 1 (Fig. 2.2(b)) is assigned to pattern node P of the LHS of PackageRule as shown by Fig. 3.3(a), then this is also a matching for PackageRule, as UML package p has no outgoing reference edges to any schemas.

Formalization of matchings

For the application of a rule we follow the single pushout approach [118] with injective morphisms. However, the definitions are slightly adapted to the proof technique of Appendix A.

Definition 16 A **matching** m **for a graph** G **in a model** M (denoted by m_G) is an injective, type conformant total morphism $m_G : G \to M$, which means that

- **Type conformance of nodes.** $\forall x \in V_G, \exists c \in V_M : t(x) \overset{*}{\leftarrow} t(c) \wedge m_G(x) = c$;
- **Type conformance of edges.** $\forall u \xrightarrow{z} v \in E_G, \exists a \xrightarrow{e} b \in E_M : t(u) \overset{*}{\leftarrow} t(a) \wedge t(v) \overset{*}{\leftarrow} t(b) \wedge t(z) = t(e) \wedge m_G(u \xrightarrow{z} v) = a \xrightarrow{e} b$;
- **Injective mapping of nodes.** $\forall x, y \in V_G : m_G(x) = m_G(y) \Longrightarrow x = y$;
- **Injective mapping of edges.** $\forall u \xrightarrow{z} v, u' \xrightarrow{z'} v' \in E_G : m_G(u \xrightarrow{z} v) = m_G(u' \xrightarrow{z'} v') \Longrightarrow z = z' \wedge u = u' \wedge v = v'$.

Definition 17 A **partial matching** m_G^{sub} **for a graph** G **in a model** M is a matching $m_{G_{sub}}$ for a subgraph G_{sub} of G in model M.

Definition 18 A **maximal partial matching** $\overset{\max}{m_G}$ **for a graph** G **in a model** M is

- a matching $m_{G_{max}}$ for a subgraph G_{max} of G in model M, provided that
- no matchings exist in model M for any such subgraphs G_{sub} of G, which also contain G_{max} as a subgraph. ($G_{max} \subseteq G_{sub} \subseteq G$)

The above definition permits several maximal partial matchings to co-exist.

Definition 19 A **matching** m **for a rule** $r = (r_b, \{(\mathsf{NAC}_i, p_{\mathsf{NAC}_i})\})$ **in a model** M (denoted by m_r) is

- a matching m for the LHS in model M, provided that
- no matching m' exists for any NAC graph, formally, $\forall \mathsf{NAC}_i, \nexists m' : \mathsf{NAC}_i \to M$, for which $\forall x \in \underline{V_{\mathsf{LHS}}} \cap V_{\mathsf{NAC}_i}, \forall x' \in \underline{V_{\mathsf{NAC}_i}} \cap V_{\mathsf{LHS}} : p_{\mathsf{NAC}_i}(x) = x' \Longrightarrow m'(x') = m(x)$, and

$$\forall u \xrightarrow{z} v \in \underline{E_{\mathsf{LHS}}} \cap E_{\mathsf{NAC}_i}, \forall u' \xrightarrow{z'} v' \in \underline{E_{\mathsf{NAC}_i}} \cap E_{\mathsf{LHS}} : \\ p_{\mathsf{NAC}_i}\left(u \xrightarrow{z} v\right) = u' \xrightarrow{z'} v' \Longrightarrow m'\left(u' \xrightarrow{z'} v'\right) = m\left(u \xrightarrow{z} v\right).$$

3.1.4 Application of graph transformation rules

The *application of a rule* to an instance model replaces a matching of the LHS in the model by an image of the RHS. This is performed in two phases.

- *Pattern matching*:
  1. find a matching of LHS in the model (by graph pattern matching),
  2. check the negative application conditions NAC, which prohibit the presence of certain objects and links
- *Updating*:
  3. remove a part of the model that can be mapped to LHS but not to RHS to produce the *context model*,

4. glue the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) to obtain the *derived model*.

A *graph transformation run* is a sequence of rule applications starting from an initial model.

Example 8 Since pattern node P of the LHS of PackageRule (Fig. 3.1(a)) can be matched to UML package p, it is applicable to Model 1 (Fig. 2.2(b)) as UML package p has not yet been transformed to a schema. In this specific case, rule application means that a new schema s with its additional reference link is added to the model. The derived instance model Model 2 is presented in Fig. 2.2(c).

Formalization of rule application

Definition 20 Given a matching m for a rule r in model M, the **deletion phase** of a rule application of the rule r is executed on a matching m in the model M yielding the context model M_c, when

- we delete all objects, to which nodes appearing only in the LHS are mapped by m, formally, $V_{M_c} = V_M \setminus \Delta V_M^-$, where $\Delta V_M^- = \{\, c \mid \exists x \in V_{\mathsf{LHS}} \setminus V_{\mathsf{RHS}} \wedge m(x) = c \,\}$; and
- we delete all links, to which edges appearing only in the LHS (but not in RHS) are mapped by m, formally,

$$\Delta E_1^- = \left\{ a \xrightarrow{e} b \mid \exists u \xrightarrow{z} v \in E_{\mathsf{LHS}} \setminus E_{\mathsf{RHS}} \wedge m(u \xrightarrow{z} v) = a \xrightarrow{e} b \right\};$$

- all dangling (i.e., incident) edges are deleted as well, formally,

$$\Delta E_2^- = \left\{ a \xrightarrow{e} b \mid \exists x \in V_{\mathsf{LHS}} \setminus V_{\mathsf{RHS}} \wedge (m(x) = a \vee m(x) = b) \right\}.$$

Deletion of links is performed as $E_{M_c} = E_M \setminus \Delta E_M^-$, where $\Delta E_M^- = \Delta E_1^- \cup \Delta E_2^-$.

Definition 21 Given a matching m for a rule r in model M, the **insertion phase** of a rule application of the rule r is executed on a matching m in the context model M_c yielding the model M', if a matching m_{RHS} can be prepared for the RHS to model M' in the following way.

- Each mapped node x' of RHS is mapped by matching m_{RHS} to the same object that has been assigned to its origin node x by matching m, formally, $\forall x' \in \underline{V_{\mathsf{RHS}}} \cap V_{\mathsf{LHS}}$, $\forall x \in \underline{V_{\mathsf{LHS}}} \cap V_{\mathsf{RHS}}$: $p(x) = x' \Longrightarrow m_{\mathsf{RHS}}(x') = m(x)$.
- For each unmapped node x' of RHS, a new object identifier c is picked and removed from the unused model element universe, then an object c of type $t(x')$ is assigned to pattern node x' by matching m_{RHS}, and finally, it is added to the set of inserted objects ΔV_M^+, formally,

$$\forall x' \in V_{\mathsf{RHS}} \setminus V_{\mathsf{LHS}},\ \exists c \in \mathfrak{U}_{NU} \ :\ c \notin \mathfrak{U}'_{NU} \wedge t(x') = t(c) \wedge m_{\mathsf{RHS}}(x') = c \wedge c \in \Delta V_M^+.$$

- When all the inserted objects are collected, they are added to model M, formally,

$$V_{M'} = V_{M_c} \cup \Delta V_M^+.$$

- Each mapped edge $u' \xrightarrow{z'} v'$ of RHS is mapped by matching m_{RHS} to the same link that has been assigned to its origin edge $u \xrightarrow{z} v$ by matching m, formally, $\forall u' \xrightarrow{z'} v' \in \underline{E_{\mathsf{RHS}}} \cap E_{\mathsf{LHS}}$, $\forall u \xrightarrow{z} v \in \underline{E_{\mathsf{LHS}}} \cap E_{\mathsf{RHS}}$: $p\left(u \xrightarrow{z} v\right) = u' \xrightarrow{z'} v' \Longrightarrow m_{\mathsf{RHS}}\left(u' \xrightarrow{z'} v'\right) = m\left(u \xrightarrow{z} v\right)$.

- For each unmapped edge $u' \xrightarrow{z'} v'$ of RHS, a new link identifier e is picked and removed from the unused model element universe, then a link $a \xrightarrow{e} b$ of type $t(z')$ is assigned to pattern edge $u' \xrightarrow{z'} v'$ by matching m_{RHS} in a source and target object preserving way, and finally, it is added to the set of inserted links ΔE_M^+, formally,

$$\forall u' \xrightarrow{z'} v' \in E_{\mathsf{RHS}} \setminus E_{\mathsf{LHS}}, \exists e \in \mathfrak{U}_{NU} :$$
$$e \notin \mathfrak{U}'_{NU} \wedge t(z') = t(e) \wedge m_{\mathsf{RHS}}\left(u' \xrightarrow{z'} v'\right) = a \xrightarrow{e} b \wedge a \xrightarrow{e} b \in \Delta E_M^+.$$

- When all the inserted links are collected, they are added to model M, formally,

$$E_{M'} = E_{M_c} \cup \Delta E_M^+.$$

Definition 22 Given a metamodel MM, and a matching m for a rule r in model M, **rule r is applied to the matching m in the model M yielding the derived model** M', (denoted by $M \overset{r,m}{\Longrightarrow} M'$) if deletion and insertion phases are executed in this order, and derived model M' is a well-formed instance of metamodel MM.

In the above definition, the well-formedness of the derived model is prescribed as a right application condition [37]. In practice, this means that effects of rule application have to be rolled back, if the derived model is not well-formed e.g., due to multiplicity constraint failure, or the existence of parallel edges of the same type.

When modeling complex systems, naturally, more than a single graph transformation rule is required. A graph transformation system encapsulates a set of rules, which can be applied during the evolution of the system model.

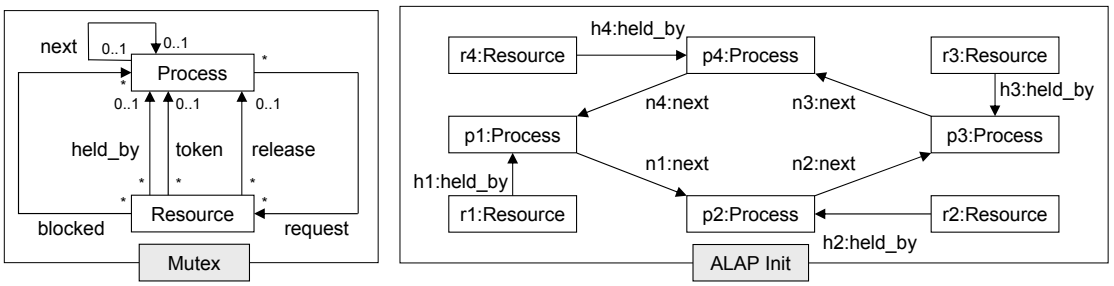
Definition 23 A **graph transformation system** $\mathfrak{S}_{GT} = (MM, R)$ is a tuple that consists of the metamodel MM, and the set of graph transformation rules R.

Definition 24 Given a graph transformation system $\mathfrak{S}_{GT} = (MM, R)$, a **graph transformation run** is a sequence of rule applications (denoted as $M_I \overset{*}{\Longrightarrow} M_n$), which starts from an initial model M_I and which applies rules from the set R.

3.2 Modeling a distributed mutual exclusion algorithm

In this section, another running example describing a distributed mutual exclusion algorithm (with full specification in [60]) is specified by graph transformation. Compared to the object-relational mapping, this algorithm represents a different application scenario, in which the dynamic semantics of a visual language is defined by means of graph transformation in an operational way. The dissimilar application criteria and environment of this algorithm and its later reuse in this thesis are arguments for introducing a second running example.

In this domain, processes try to access shared resources. One requirement of the algorithm is that each resource may be accessed by at most one process at a time. This is achieved by using a token ring of processes. In the consecutive phases of the algorithm, a process may issue a request on a resource, the resource may eventually be held by a process, and finally, a process may release the resource. The right to access a resource is modeled by a token. The algorithm also contains a deadlock detection procedure, which has to track the processes that are blocked.



(a) Metamodel for the mutual exclusion problem

(b) A sample instance model

Figure 3.4: Metamodel and model for the mutual exclusion problem

3.2.1 Metamodels and instance models

The metamodel of the mutual exclusion problem is depicted in Fig. 3.4(a). It has only two classes: Process and Resource. Links of type next organize processes into a ring. The access right of a process to a resource is symbolized by the association token. Associations release or request show if a process releases or requests for a resource, respectively. A resource may also be held_by a process. Links of type blocked express if processes are directly or indirectly blocked by an already used resource. No inheritance is specified in the figure.

A well-formed instance model of this domain is shown in Fig. 3.4(b). It has four processes (p1 to p4) and four links (n1 to n4) of type next, which organize processes into a ring. Four resources (r1 to r4) also appear in the model. Each resource is held by a separate process, which can be expressed by the four edges of type held_by (h1 to h4) connecting the resources to the corresponding processes. Furthermore, the instance model of Fig. 3.4(b) obviously satisfies all multiplicity constraints of the metamodel.

3.2.2 Graph transformation rules

The distribute mutual exclusion algorithm can be described by 13 graph transformation rules as presented in Fig. 3.5.

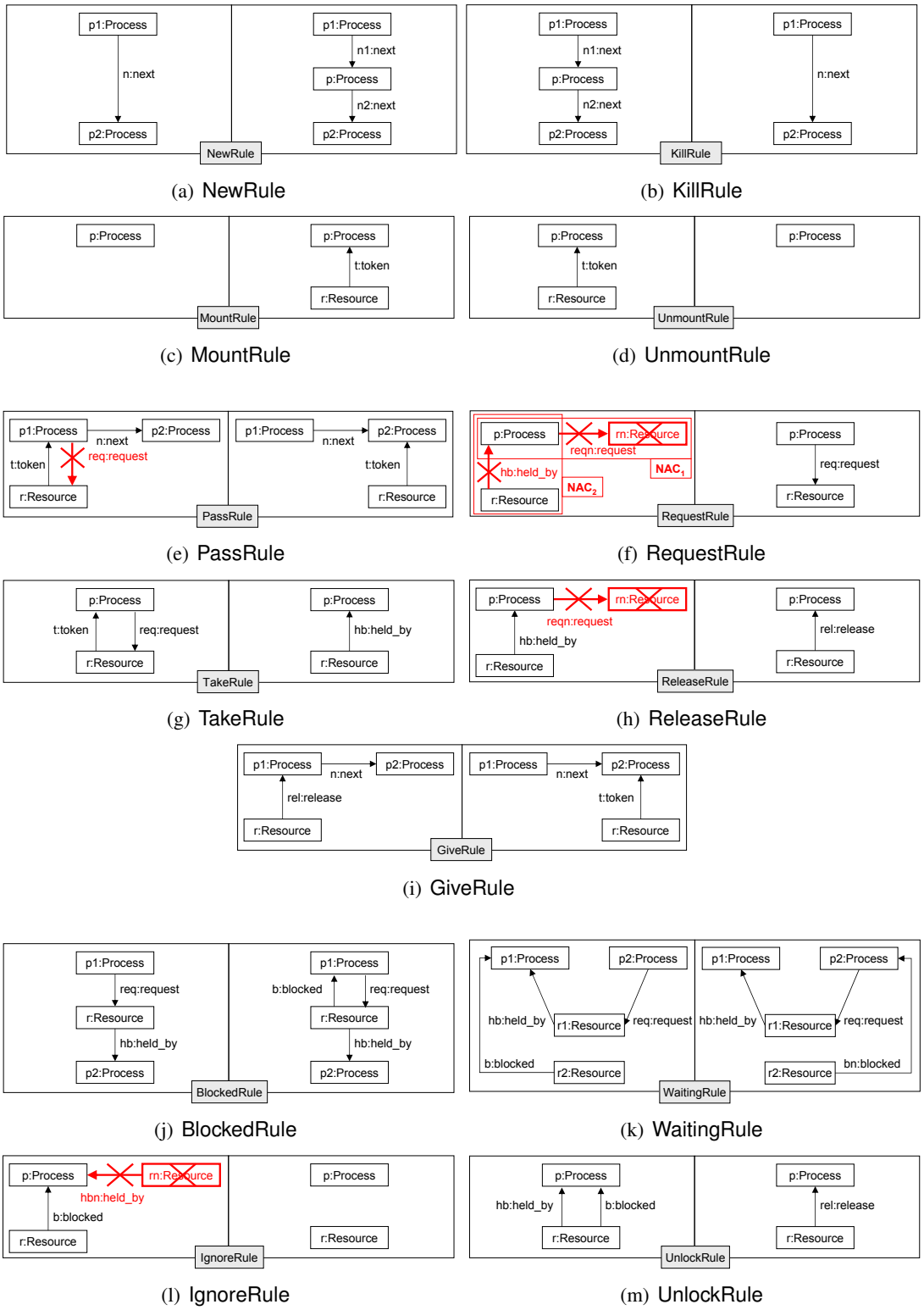
(a) NewRule (Fig. 3.5(a)) searches for two consecutive processes p1 and p2 in the token ring, and places a new process p in between p1 and p2.

(b) KillRule (Fig. 3.5(b)) searches for three consecutive processes p1, p, and p2 in the token ring, and removes process p from the middle.

(c) MountRule (Fig. 3.5(c)) seeks for a process p, creates a new resource r, and gives the initial access right (modeled by a token link) of resource r to process p.

(d) UnmountRule (Fig. 3.5(d)) tries to find a process p, which has a token for resource r, and removes the resource from the system.

(e) PassRule (Fig. 3.5(e)) gives the access right of process p1 for resource r to the following process p2 in the token ring, if process p1 has not yet issued a request for resource r.



(a) NewRule (b) KillRule

(c) MountRule (d) UnmountRule

(e) PassRule (f) RequestRule

(g) TakeRule (h) ReleaseRule

(i) GiveRule

(j) BlockedRule (k) WaitingRule

(l) IgnoreRule (m) UnlockRule

Figure 3.5: Rules describing the mutual exclusion algorithm

(f) RequestRule (Fig. 3.5(f)) expresses that if resource r is not yet held by process p, and process p has not issued a request for any other resources, then it requires resource r.

(g) TakeRule (Fig. 3.5(g)) gives access to a resource r for process p (expressed by a held_by edge from r to p), if process p requested for such a resource r, for which it already had access right (i.e., a token) as well.

(h) ReleaseRule (Fig. 3.5(h)) replaces the held_by edge connecting resource r to process p by a release edge, if process p has no requests for any resources.

(i) GiveRule (Fig. 3.5(i)) passes the token for a resource r released by process p1 to the next process p2 in the token ring.

(j) BlockedRule (Fig. 3.5(j)) expresses that if process p1 requests for a resource r already held by another process p2, then process p1 gets blocked by resource r.

(k) WaitingRule (Fig. 3.5(k)) propagates the blocking of resource r2 from process p1 to process p2, if p2 requests for a resource r1 held by process p1.

(l) IgnoreRule (Fig. 3.5(l)) liberates the blocking of resource r on process p, if process p has no access to any resources.

(m) UnlockRule (Fig. 3.5(m)) detects a deadlock at process p, which has access to a resource r, which causes a blocking for the same process p. UnlockRule unlocks the circular blocking by forcing process p to release resource r.

For example, ReleaseRule (Fig. 3.5(h)) can be applied on the model that has been presented in Fig. 3.4(b). Let us suppose that in the pattern matching phase, p, hb and r of the precondition of ReleaseRule are mapped to p1, h1 and r1 of the model, respectively. Since the selected process p1 does not have any associated requests, the negative application condition does not prohibit the execution of the rule. In the updating phase, edge h1 is removed from the model, and a new edge rel1 of type release is created.

3.3 Graph transformation tools

In the beginning of my research (in 2003), the following graph transformation tools, which can be considered as successors of the first visual graph transformation environment of [54], represented the state of the art, which are listed in alphabetical order.

- AGG [42] is a development environment for attributed graph transformation systems supporting an algebraic approach to graph transformation. It aims at specifying and rapid prototyping applications with complex, graph structured data. The main strengths of AGG are its analysis techniques, namely, the critical pair analysis [61] and consistency checking [62].

  AGG may also be used as a general purpose, interpreted graph transformation engine, which has a category theory based implementation written in Java. Its algorithm [119] interprets pattern matching as a constraint satisfaction problem. AGG runs in main memory without using an underlying database.

- AToM3 [31] is a tool for multi-paradigm modelling. Starting from metamodel specifications, AToM3 can generate graphical tools for manipulating well-formed models. In addition, model transformation is supported by means of graph transformation, whose rules can be specified declaratively in the framework.

  AToM3 has an interpreted, in-memory pattern matching engine written in Python, which uses a local search based technique.

- BOTL [91] is a specification language and mechanism for the bidirectional transformation of object-oriented models. Specification is based on the graph transformation concept, which is complemented by algorithmic expressions, which relate object identities and attributes.

  Tool support for BOTL is stated as an ongoing work in [21], so exact details of its pattern matching algorithm are unknown.

- DiaGen [96] is a rapid prototyping tool for developing powerful diagram editors. It provides support for specifying, editing, and analyzing diagrams, and for automatically generating visual editors from diagram specifications, which are represented by graph transformation rules.

  The main strength of DiaGen is its structural analysis module, which can validate syntactic correctness of diagrams on-line during the editing process. This module is based on interpreted hypergraph transformation and it is supplemented by syntax highlighting and an interactive layout facility to visualize the result of structural analysis.

- FUJABA [77] is an open source UML CASE tool, which focuses on the rule-based development of Java applications with complex data structures. The envisioned development process with FUJABA starts from specifications described by UML diagrams. In order to define the exact behaviour of the system under design, UML diagrams are supplemented by further graphical data structures such as graph transformation rules and story diagrams [44], which latter describes control flow. These specifications are then used to guide the automatic code generation process, which produces source code in Java.

  FUJABA uses an in-memory, local search technique for pattern matching, and it belongs to the group of compiled graph transformation tools.

- GREAT [1] is a language for graph rewriting and transformations, and a corresponding tool for building model transformation applications inside the Generic Modeling Environment [83] (GME) framework, which aims at creating domain-specific modeling and program synthesis environments.

  GREAT has a pattern specification, a graph transformation and a control flow language, which define the behaviour of the pattern matcher, the rule executor, and the sequencer module, respectively. This tool performs uni-directional transformation. Its pattern matching engine [158] is a local search based approach, and it can run both in interpreted and in compiled mode, in which C++ code is generated. The pattern matcher 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 terminology.

- Groove [113] is a graph transformation based model checking approach. In this sense, graph based models represent the states of the transition system, and graph transformation is used for specifying transitions. Groove can be used for modeling the design-time, compile-time, and run-time structure of object-oriented systems, and its main benefit is the ability to verify model

transformation and dynamic semantics through an (automatic) analysis of the resulting graph transformation system.

From a technical point of view, Groove generates full state space by exhaustively applying all enabled GT rules at each state. Each newly generated state is compared to all other states up to isomorphism. The pattern matching module runs in interpreted mode and performs a local search based algorithm.

- PROGRES [122] is an integrated set of tools, which supports programming with graph transformation. It has a graph-oriented data model and a hybrid (textual and graphical) syntax for specifying dynamic behaviour, which are backed by a syntax-directed graphical editor.

  PROGRES operates on an underlying graph based database (GRAS) [76], and it has an integrated interpreter, which translates specifications into intermediate code and executes this code afterwards. PROGRES can also run in compiled mode as both C and Java code can be generated from the specifications. This tool uses a local search based strategy for pattern matching.

- VIATRA2 [6] is a transformation-based verification and validation framework, which aims at improving the quality of systems designed within the Unified Modeling Language by automatically checking consistency, completeness, and dependability requirements like in case of [143]. This feature is supported by automatically executable, provenly correct and complete model transformations [142, 144], which are specified by the mathematically precise rule-based specification formalisms of graph transformation (GT) and Abstract State Machines (ASM).

  The graph pattern matching module of VIATRA2 is an interpreted engine. Its earlier version provided a CSP-based technique and it was written in Prolog, while the current version is written in Java, and it performs local search in the pattern matching phase.

- VMTS [85] is a graphical metamodeling environment, which again enables model editing and graph transformation in a single tool. The main strength of this tool is its full feature support for validating OCL constraints [84]. These constraints are attached to the specification of graph transformation rules, and they are checked at run-time by aspect-oriented constraint management techniques.

  VMTS is an interpreted approach as graph transformation is executed by the Visual Model Processor module, and pattern matching is performed by a local search approach.

The recently developed graph transformation tools are the following.

- Graph Rewrite Generator (GrGen) [51] is a generative programming system for graph transformation, which originally aimed at finding patterns in graph-based intermediate representations being used in compiler construction. GrGen represents models as uses attributed, typed, and directed multigraphs with multiple inheritance on node and edge types.

  GrGen has been recently reimplemented for the .NET framework. This new version generates .NET assemblies by a generator written in Java, and it additionally contains a graph backend written in C#.

- MOLA [69, 70] is a graphical and procedural model transformation language together with a tool consisting of a Transformation Definition Environment and a Transformation Execution Environment. Both environments use an underlying RDBMS based repository for storing models, metamodels and transformations.

- TefKat [82] implements a state-of-the-art declarative model transformation language suitable for MDD. It is implemented as an Eclipse plugin that leverages the Eclipse Modeling Framework (EMF) to handle models based on MOF, UML2, and XML Schema. Tefkat is specifically designed for writing scalable and re-usable transformation specifications using high-level domain concepts.

  The transformation module of TefKat uses a search plan driven technique, in which operations can be re-ordered based on efficiency and semantic correctness criteria.

3.4 Graph transformations tool representatives

We selected four graph transformation tools to participate in our analysis in later chapters. Our primary aim in selecting tools was to include those with essentially different pattern matching strategies and heterogeneous execution environments.

- AGG is an interpreted graph transformation tool written in Java, which directly follows a category theory based implementation. It specifies pattern matching as a constraint satisfaction problem to be solved. AGG runs in main memory without using any underlying databases.
- We selected the compiled version of PROGRES to run during performance measurements in the form of C programs. PROGRES uses a local search based strategy for pattern matching, and it operates on an underlying graph based database (GRAS) [76].
- FUJABA is a compiled GT tool, which performs the transformations as Java programs. FUJABA operates in the main memory, and it uses a local search based technique for pattern matching.
- The database (DB) approach, which is going to be presented in Chapter 6 as a contribution of this thesis, operates on a standard relational database by issuing join based queries for pattern matching, which categorizes this approach among the interpreted graph transformation tools. It communicates with the database via the standard JDBC interface.

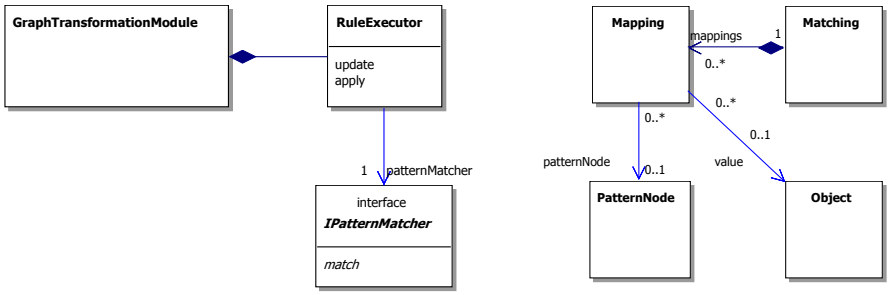
3.5 Implementation of graph transformations

The typical architecture and the basic data structures of a graph transformation module implementation are presented in Figures 3.6(a) and 3.6(b), respectively, and obtained by the analysis and the generalization of existing approaches.

At compile-time, a *graph transformation module* is built by using a graph transformation system specification. For each rule in the specification, a corresponding *rule executor* is prepared, which provides rule application functionality via its `apply()` method. Each rule executor has a *pattern matcher*, which provides pattern matching functionality via its `match()` method for the precondition of the rule, for which the corresponding executor has been generated.

A graph transformation module uses the additional data structure of `Matching`s during its operation. A `Matching` consists of `Mapping`s, which can be considered as `PatternNode-Object` pairs. Note that in the implementation, only nodes of the patterns and their corresponding matched objects are stored explicitly in a `Matching`, while the mappings of edges are omitted from this data structure as links do not have identities according to our assumptions.

At runtime, rule application is initiated by selecting a rule and invoking the `apply()` method of its executor. An initial (partial) matching is also passed as input parameter. The behaviour of the `apply()` method of a rule executor is presented by the sequence diagram of Fig. 3.7.



(a) The structure of a graph transformation engine module (b) Basic data structures

Figure 3.6: Typical data structures of a graph transformation engine

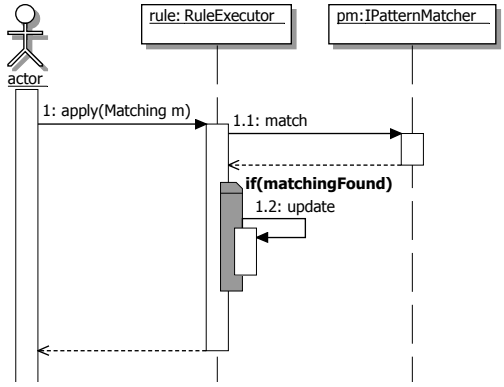


Figure 3.7: Sequence diagram for the `apply()` method

As it can be seen, the `apply()` method shows a one-to-one correspondance to the theory based definition of rule application. In this sense, activities correspond to the following two phases.

- **Pattern matching.** The `match()` method of the pattern matcher is invoked with the initial matching as input parameter. This method completes the matching by adding appropriate mappings for initially unmatched pattern nodes.

- **Updating.** If a complete matching has been found in the pattern matching phase, the `update()` method of the rule executor is invoked for handling the tasks of deletion and insertion phases.

As techniques for the implementation of the `match()` method are going to be discussed in details in all the upcoming chapters, sample Java codes that implement this method can be found at several locations (e.g., Listings 4.1 and 4.2 in Section 4.2.6). A Java program that describes the manipulation of models has already been presented by Listing 2.2.

3.6 Conclusion

In this chapter, by using the object-relational mapping example, the paradigm of graph transformation has been presented as a rule-based specification language for manipulating graph models. In addition, a distributed mutual exclusion algorithm has been introduced to represent another application scenario of graph transformation by providing a way to define the dynamic semantics of a visual language. Then an overview has been given on state-of-the-art graph transformation tools, of which representatives have been selected for our later performance measurements. Finally, a graph transformation module implementation has been discussed by presenting its typical architecture and its basic data structures.

CHAPTER

4 Pattern Matching Strategies

This chapter serves as a framework of existing, graph transformation related concepts, techniques, and heuristics, which is going to be used later for positioning the new results of this thesis. A skeletal, general purpose graph pattern matching algorithm is presented, into which all the heuristics used by both current and upcoming tools can be plugged. By analyzing this algorithm, complexity considerations of pattern matching are examined both from a theoretical and practical viewpoint. Finally, the widely used concept of search plan driven pattern matching is presented, which can be used for describing heuristics.

4.1 A general purpose graph pattern matching algorithm

As a result of intensive research, several graph pattern matching algorithms have been developed during the last decades. Some publications [25, 32, 46, 56, 89, 90, 92, 117, 131] proposed efficient pattern matching techniques for restricted classes of graphs, while others developed algorithms [29, 81, 140, 141] for graphs without structural restrictions. In the graph transformation community, variants of Ullmann [140] and VF2 [29] algorithms are used most frequently. In Algorithm 4.1, a skeleton is presented to demonstrate the typical structure of all relevant pattern matching algorithms.

The pattern matching algorithm consists of a single recursive procedure `match`(k, m) which gets the recursion level k and a matching m as its inputs. Procedure `match`(k, m) is initially invoked at recursion level 1 with a partial matching m, which is specified outside the pattern matcher by the user, and which contains mappings for a subset of pattern nodes. Since the procedure already tries to find a mapping for the first unmapped pattern node at recursion level 1, it should also be checked in the beginning by $check(0, m)$ whether the initial partial matching m represents a graph morphism.

If matching m is complete, then it can be returned as a solution (Lines 1–2). If matching m is not yet complete (Lines 3–11), then attempts are made to extend the matching. For this reason, a set of mapping candidates $P(k, m)$ is computed (Line 4), and then each candidate (n, o), which represents the mapping of pattern node n of LHS (or NAC) to object o, is added to matching m resulting in a matching m' (Line 6). If this new matching m' passes all the tests prescribed by $check(k, m')$ (Line 7), the procedure `match()` can be invoked recursively in Line 8 with parameters $k + 1$ and matching m'.

Algorithm 4.1 The skeletal pattern matching algorithm `match`(k, m)

PROCEDURE `match`(k, m) {k is the recursion level, and m is the initial partial matching}
Initially: $k = 1$ and $check(0, m) = true$ {The algorithm is initially invoked at recursion level 1, and it checks whether pattern edges that connect pattern nodes contained by the initial partial matching m can be mapped to links in the model.}
1: **if** m represents a total morphism from LHS to model M **then**
2: **return** m
3: **else**
4: Compute the set of mapping candidates $P(k, m)$
5: **for all** $(n, o) \in P(k, m)$ **do**
6: $m' := m \cup (n, o)$ {Compute the morphism m' obtained by adding (n, o) to m}
7: **if** $check(k, m')$ {Verifies whether m' is a morphism} **then**
8: **return** `match`$(k + 1, m')$
9: **end if**
10: **end for**
11: **end if**

Implementations of the pattern matching algorithm (in different GT tools) typically differ from each other in the technique of computing mapping candidates and checking extended matchings in Lines 4 and 7, respectively. In this sense, the above skeletal algorithm provides a *uniform description* for all the existing and new pattern matching strategies and heuristics. In order to be able to analyze these algorithm variants, we need an appropriate formalism for describing the search space being traversed by Algorithm 4.1 during pattern matching.

4.1.1 Search space tree

For this reason, a *snapshot* is constructed from the input parameters (i.e., recursion depth level k and matching m), whenever the `match()` method is invoked. These snapshots are then organized into a search space tree by also taking into account the method invocation hierarchy of Line 8.

A *search space tree (SST)* is a tree having snapshots as its nodes. The root of the tree is on the 0th level of recursion and it corresponds to the initial snapshot that has been prepared, when the `match()` method is invoked from outside the pattern matcher (i.e., from the rule executor) with recursion level 1 and with the initial partial matching. A snapshot node s' consisting of recursion level $k + 1$ and a (partial) matching m' appears on the kth level of the search space tree as a child of snapshot node s representing recursion level k and (partial) matching m, if m' has been obtained from m by executing Line 6 on the kth recursion level at some time during pattern matching. Consequently, if a pattern has l nodes to be matched ($l \leq |V_{\mathsf{LHS}}|$), then the search space tree has at most $l + 1$ levels, and only nodes on the lth level may denote complete matchings for the pattern.

Example 9 A sample search space tree is depicted by Fig. 4.1. This tree can be generated by a pattern matching process, which tries to search for matchings for GiveRule of Fig. 3.5(i) in the instance model of Fig. 3.4(b) by seeking mappings for pattern nodes in the P1, P2, R order.

In this case, the `match()` method has been invoked with the empty matching as it is shown by the root of the tree. On the second level, such snapshots can be found, which correspond to matchings, in which only pattern node P1 has been mapped. As it is shown by Fig. 4.1, pattern node P1 is mapped to processes p1, p2, p3, and p4. Snapshots on the third level represent matchings, in which pattern nodes P1, and P2 are already mapped. For each process assigned to pattern node P1, there is exactly one

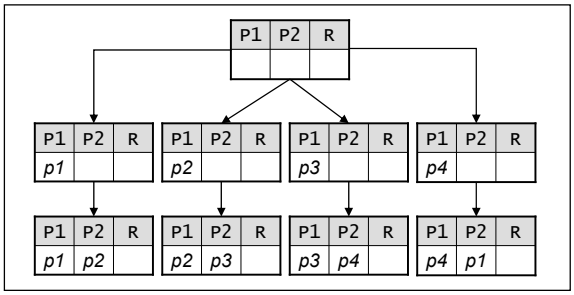


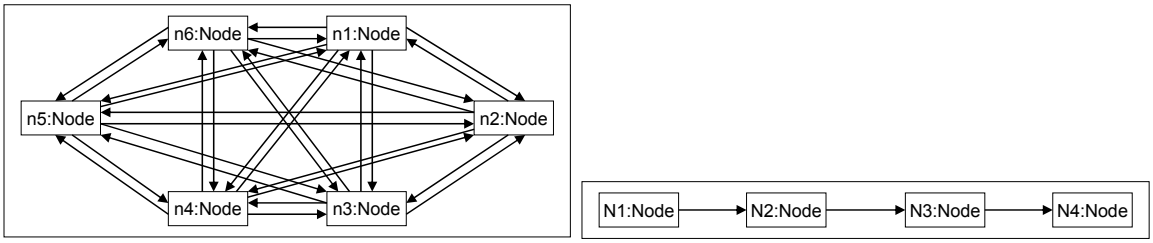
Figure 4.1: A sample search space tree

following process in the ring that can be a mapping of pattern node P2. This is reflected in Fig. 4.1 by the fact that each snapshot on the second level has exactly one child.

4.1.2 Complexity analysis of pattern matching and updating phases

From a theoretical viewpoint, no fast algorithms can be guaranteed to exist for graph pattern matching, as it leads to the subgraph isomorphism problem, for which NP-completeness has been proved [5]. The exponential worst-case complexity of the pattern matching phase can be easily demonstrated by the following example.

Example 10 Let us suppose that (i) the metamodel has a single node and edge type, (ii) the instance model is a directed complete graph, in which each pair of nodes is bidirectionally connected to each other (as in Fig. 4.2(a)), and (iii) the LHS pattern is a path graph (like the one in Fig. 4.2(b)). In such a situation, if *all the matchings* are aimed to be listed, then even this enumeration requires an exponential number of steps, and no analysis and heuristics can help to speed-up the pattern matching process.



(a) An instance model: a complete graph with 6 objects (b) An LHS pattern: a path graph with 4 nodes

Figure 4.2: Illustrative example for the complexity analysis of pattern matching

If injectivity checking is omitted from the pattern matching process and the evaluation order of LHS nodes is fixed (e.g., to a left-to-right order in this case), the size of the SST can be estimated as follows. The leftmost node of the LHS can be mapped to $|V_M|$ objects. For each such mapping, the second node of the LHS can be independently fixed to $|V_M|-1$ objects, and this argument can be repeated for all the remaining nodes of the LHS. As a consequence, the size of the SST is in $\mathcal{O}\left(|V_M|^{|V_{\mathsf{LHS}}|}\right)$, and each leaf of the SST denotes a matching for the LHS pattern in the instance model. The situation is even worse,

if nodes of the LHS are allowed to be processed in an arbitrary order, since this gives an additional $|V_{\mathsf{LHS}}|!$ factor for the size of the SST as each matching is enumerated that many times.

Fortunately, in practical model transformation problems of software engineering, search space can be reduced due to several reasons.

- The size of LHS graphs are typically constant except for some rather exotic approaches like shaped hypergraph transformations. This makes the time complexity to be bound by a polynomial, in which the exponent is also constant but not necessarily small.
- Instance models from software engineering domains are always sparser than the one in Fig. 4.2(a). In a typical situation, an object is connected via links to a couple of other objects, but not to all objects in the model. As a consequence, if each object stores links to its neighbourhood and navigation is always performed only to the neighbours of a given object, this can reduce the search space significantly by the ratio of neighbouring objects and all objects.
- Metamodels typically contain more than one class and association, which means that only type conformant objects and links have to be enumerated when a matching is being extended. In a typical model repository, neighbours of an object are stored separately according to the type of the connecting link. As a consequence, if navigation is performed only along links of a given type, this can further cut the search space by simply omitting such neighbours that can only be reached via a link of a different type.
- Instance models are typically less regular than the one in Fig. 4.2(a). Irregularities in instance models and LHS patterns increase the gap between the size of search spaces that are traversed according to different LHS node orders. This opens up the way to construct heuristics to reduce the search space by fixing a good (or ideally an optimal) LHS node evaluation order for pattern matching. Note that finding such an order constitutes a highly critical part in the process of pattern matching.
- A fifth source of state space reduction stems from injectivity checking, which disallows different LHS nodes of the same type to be mapped to the same object in the instance model.

By using the above techniques for typical model transformation problems from the software engineering industry, the size of the SST (i.e., the "practical complexity" of graph pattern matching) can be reduced to a scale, which can be overapproximated by a linear or quadratic function of the model size in many applications.

The complexity of the updating phase is linear in the size of the graph transformation rule (i.e., $\mathcal{O}\left(|V_{\mathsf{LHS}}| + |V_{\mathsf{RHS}}| + |E_{\mathsf{LHS}}| + |E_{\mathsf{RHS}}|\right)$), even if only a rough estimate is used, as at most that many objects and links can be deleted or created.

As a consequence of this short analysis, the rest of this thesis focuses on the performance issues of the pattern matcher as it has significantly larger influence on the overall behaviour and performance of the graph transformation module. This huge gap between the significance of the two phases is also reflected by the detailedness of discussions on pattern maching and updating phases in each of the upcoming chapters.

It is worth emphasizing that the above-mentioned theoretical complexity analysis does not take into account several factors, which might cause significant performance degradation in practice, which obviously affects the measurement results. These factors include the tasks related to querying and updating indexes or to the administration of large number of objects and links, which might influence both the pattern matching and the updating phase.

4.2 Search plan driven pattern matching

The generation of search plans [163] is a frequently used and efficient strategy to drive the execution of (local search based) pattern matching algorithms. Informally, a search plan defines an order of pattern nodes, in which they are bound to objects of the instance model during pattern matching. In addition to simply specifying the binding order of pattern nodes, it often also includes an order of elementary operations that have to be executed to drive pattern matching. In the context of Algorithm 4.1, this latter task means the appropriate (and recursion level dependent) definition of mapping candidate computation ($P(k, m)$ in Line 4) and matching checks ($check(k, m')$ in Line 7).

In this thesis, the process of search plan driven pattern matching is as follows. At compilation time, a so-called search graph is constructed for each LHS and NAC patterns. At runtime, when bindings of the input partial matching are known, the search graph is complemented by an adornment (i.e., a binding pattern), which denotes if a given pattern node is initially bound or free. In the second step, a search plan is generated from the adorned search graph. Finally, the search plan is used to drive the pattern matching process.

In case of compiled graph transformation approaches, the overall process slightly differs as search plans are usually prepared for all combinations of binding and pattern matching code fragments are also generated and compiled for all search plans at compile-time. In this case, only the selection and the execution of the appropriate code fragment remain a runtime task.

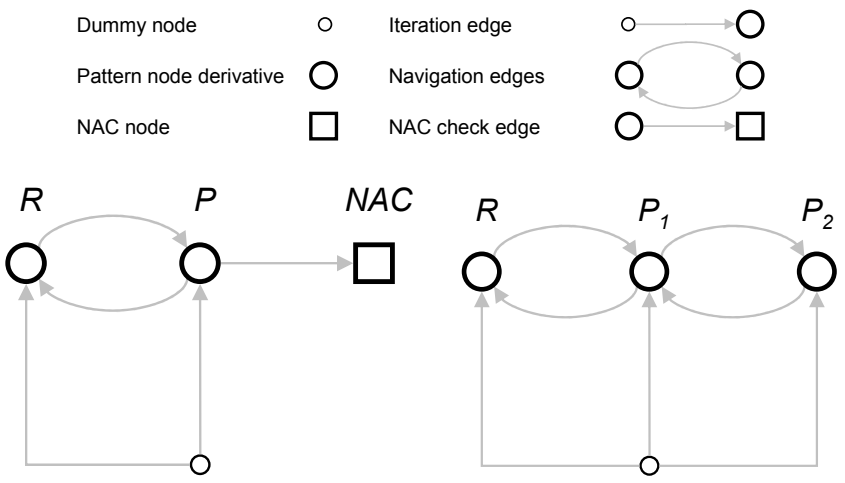
4.2.1 Search graphs

A *search graph* is a directed graph generated for each pattern specification.

(a) A search graph contains a single *dummy node* (denoted by a small circle) that represents a possible starting point for any search graph traversals.
(b) Each node of the pattern is mapped to a *pattern node derivative* (denoted by a large circle) in the search graph.
(c) For each negative application condition of the pattern, a *NAC node* (denoted by a large rectangle) is also added.
(d) *Iteration edges* are directed edges connecting the dummy node to every pattern node derivative.
(e) Each navigable direction of each pattern edge is mapped to a *navigation edge* in the search graph. In a typical case, when a pattern edge is navigable in both directions, a pair of navigation edges with end nodes connected in both directions is created for each such pattern edge.
(f) For each node shared between the pattern and the embedded negative application condition, a *NAC check edge*, which connects the pattern node derivative to the NAC node is added to the search graph.

A *to-one (to-many) navigation edge* represents a pattern edge navigation in a direction, in which the navigation target has an at most one (arbitrary) multiplicity constraint. To-one and to-many navigation edges are denoted by edges with labels 0..1 and *, respectively.

Example 11 Search graphs describing LHS patterns of graph transformation rules ReleaseRule of Fig. 3.5(h) and GiveRule of Fig. 3.5(i) are depicted in Fig. 4.3(a) and 4.3(b), respectively. Note that not only nodes and edges of the LHS, but also the negative application conditions (e.g., NAC in Fig. 4.3(a)) should be represented in the search graph.



(a) ReleaseRule (b) GiveRule

Figure 4.3: Search graphs for the preconditions of different rules

Formalization of search graphs

Definition 25 Given a metamodel MM and a graph transformation rule r, the **search graph** $\mathsf{SG} = (V_{\mathsf{SG}}, E_{\mathsf{SG}}, b)$ **of the** LHS **pattern** is a directed graph with nodes V_{SG} and edges E_{SG}, and a backward mapping function (graph morphism) $b : \mathsf{SG} \rightarrow r_{\mathsf{PRE}}$, which maps nodes and edges of search graph SG to the preconditions r_{PRE} of rule r (see Definition 12). The structure of search graphs can be described by the following rules:

- Nodes V_{SG} of the search graph can be partitioned into (i) the singleton set $\{\,\mathsf{d}\,\}$ containing a dummy node d, (ii) pattern node derivatives V^P_{SG}, and (iii) NAC nodes $V^{\mathsf{NAC}}_{\mathsf{SG}}$, formally, $V_{\mathsf{SG}} = \{\,\mathsf{d}\,\} \cup V^P_{\mathsf{SG}} \cup V^{\mathsf{NAC}}_{\mathsf{SG}}$, $\mathsf{d} \notin V^P_{\mathsf{SG}}$, $\mathsf{d} \notin V^{\mathsf{NAC}}_{\mathsf{SG}}$, and $V^P_{\mathsf{SG}} \cap V^{\mathsf{NAC}}_{\mathsf{SG}} = \emptyset$. Mapping rules for nodes of the search graph are as follows:
  - A **dummy node** d (denoted by a small circle) is a node of the search graph without an origin in the LHS graph, and it represents a possible starting point for any search graph traversals. Formally, $\exists \mathsf{d} \in V_{\mathsf{SG}}, \forall x \in V_{\mathsf{LHS}} : b(\mathsf{d}) \neq x$.
  - Each node x of the LHS pattern is mapped to a **pattern node derivative** x (denoted by a large circle) in the search graph. Formally, $\forall x \in V_{\mathsf{LHS}}, \exists \mathsf{x} \in V^P_{\mathsf{SG}} : b(\mathsf{x}) = x$.
  - Each negative application condition NAC_i of the LHS pattern is mapped to a NAC **node** n_i (denoted by a large rectangle) in the search graph, i.e., each NAC is represented by a single "proxy" node. Formally, $\forall \mathsf{NAC}_i, \exists \mathsf{n}_i \in V^{\mathsf{NAC}}_{\mathsf{SG}} : b(\mathsf{n}_i) = \mathsf{NAC}_i$.
- Edges E_{SG} of the search graph can be partitioned into (i) iteration edges E^{iter}_{SG}, (ii) navigation edges E^{nav}_{SG}, and (iii) NAC check edges $E^{\mathsf{NAC}}_{\mathsf{SG}}$, formally, $E_{\mathsf{SG}} = E^{iter}_{\mathsf{SG}} \cup E^{nav}_{\mathsf{SG}} \cup E^{\mathsf{NAC}}_{\mathsf{SG}}$, $E^{iter}_{\mathsf{SG}} \cap E^{nav}_{\mathsf{SG}} = E^{iter}_{\mathsf{SG}} \cap E^{\mathsf{NAC}}_{\mathsf{SG}} = E^{nav}_{\mathsf{SG}} \cap E^{\mathsf{NAC}}_{\mathsf{SG}} = \emptyset$. Mapping rules for edges of the search graph are as follows:
  - The dummy node d is connected to each pattern node derivative x by an **iteration edge** $\mathsf{d} \xrightarrow{\mathsf{i}} \mathsf{x}$ without an origin in the LHS pattern. Formally, $\forall \mathsf{x} \in V^P_{\mathsf{SG}}, \exists! \mathsf{d} \xrightarrow{\mathsf{i}} \mathsf{x} \in E^{iter}_{\mathsf{SG}}$.

– Each pattern edge $u \xrightarrow{z} v$ of the LHS pattern connecting node u to node v is mapped to a pair of **navigation edges** connecting corresponding end node derivatives u and v in both directions (i.e., $\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v}$ and $\mathsf{v} \xrightarrow{\mathsf{z}_{inv}} \mathsf{u}$). Formally,

$$\forall u \xrightarrow{z} v \in E_{\mathsf{LHS}},\ \exists \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E^{nav}_{\mathsf{SG}},\ \exists \mathsf{v} \xrightarrow{\mathsf{z}_{inv}} \mathsf{u} \in E^{nav}_{\mathsf{SG}} :$$
$$\left(b\left(\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v}\right) = u \xrightarrow{z} v \wedge b\left(\mathsf{v} \xrightarrow{\mathsf{z}_{inv}} \mathsf{u}\right) = u \xrightarrow{z} v\right).$$

– For each node u of the LHS graph that is shared with the negative application condition pattern NAC_i, a NAC **check edge** $\mathsf{u} \xrightarrow{\mathsf{z}_i} \mathsf{n}_i$ connecting the corresponding pattern node derivative u to NAC node n_i is added to the search graph. Formally, $\forall \mathsf{NAC}_i,\ \forall u \in \underline{V_{\mathsf{LHS}}} \cap V_{\mathsf{NAC}_i},\ \exists \mathsf{u} \xrightarrow{\mathsf{z}_i} \mathsf{n}_i \in E^{\mathsf{NAC}}_{\mathsf{SG}} : b(\mathsf{u}) = u \wedge b(\mathsf{n}_i) = \mathsf{NAC}_i$.

By using a similar construction, a search graph can be built for each negative application condition itself. This way negative application conditions embedded into each other at an arbitrary depth can be handled [114].

4.2.2 Adorned search graphs and search plans

An *adorned search graph* is a search graph, in which pattern node derivatives are adorned depending on the initial binding of their pattern node origin. The set of derivatives that represent pattern nodes that are already matched when the pattern matching starts are called *bound nodes (B)* and denoted by circles surrounded with dashed boxes. The remaining (initially unmatched) pattern node derivatives are called *free nodes (F)* and denoted by numbered circles.

A *search plan* is a traversal of such spanning trees of the adorned search graph that are rooted at some bound nodes or at the dummy node. A traversal defines a sequence in which edges are traversed. The position of a given edge in this sequence and the position of a free node in the corresponding binding order are marked by increasing integers written on the black edges of spanning trees and inside free nodes, respectively, as in Figs. 4.4(a) to 4.4(d). (Note that the number appearing on a search plan edge should always and inherently be the same as the integer value located inside its target node.)

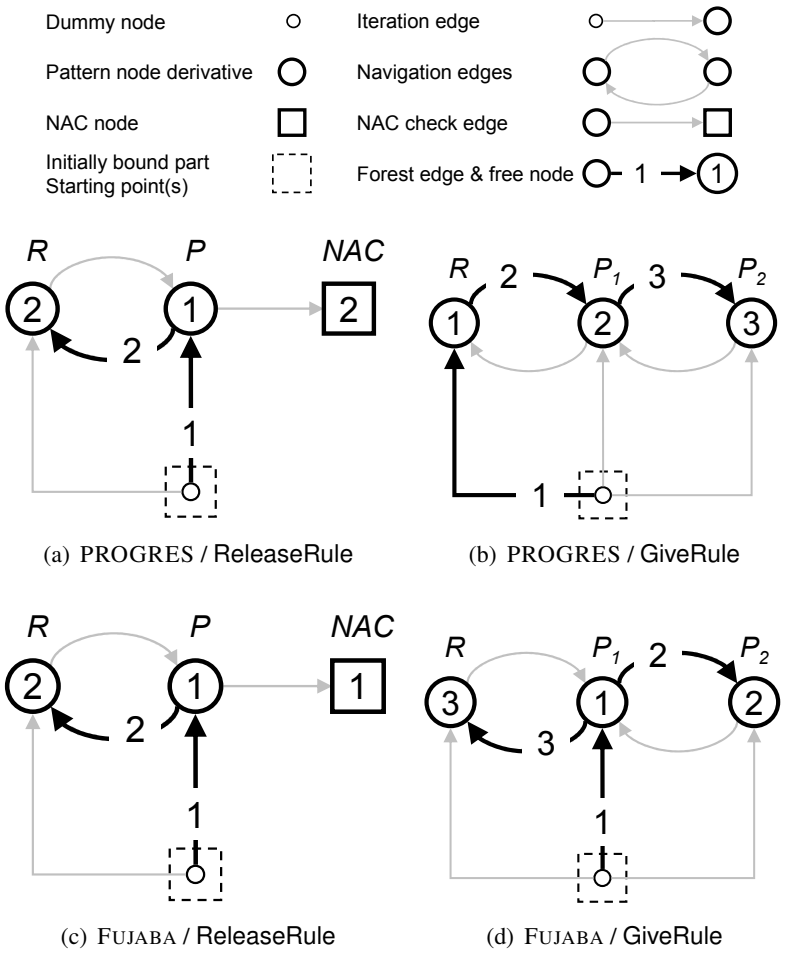
Example 12 Search plans being generated for the preconditions of ReleaseRule and GiveRule by PROGRES and FUJABA are depicted in Fig. 4.4. It should be emphasized that these search plans are directly derived from the generated (Java or C) source code as a result of manual code analysis, and multiplicity constraints have not been considered during the generation of these search plans.

For example, the search plan of Fig. 4.4(b) has been generated for the case, when all input parameters are free at the time of the pattern invocation. The spanning tree is denoted by the thick black lines, and it is rooted at the dummy node. The search plan specifies that pattern nodes have to be mapped in the R, P1, P2 order by traversing first the iteration edge leading to derivative R, then the navigation edge connecting derivatives R to P1, and finally, the navigation edge linking derivatives P1 to P2.

By comparing Figures 4.4(a) and 4.4(c), it can be seen that PROGRES checks the negative application condition at the end of the pattern matching process, while FUJABA performs the same check as soon as pattern node P is already mapped.

4.2.3 Formalization of adorned search graphs and search plans

Definition 26 An **adorned search graph** ASG is a run-time representation of a search graph SG, in which pattern node derivatives are further partitioned into **bound nodes** V^B_{SG} and **free nodes** V^F_{SG} depending on whether the corresponding node of the LHS pattern is already bound when pattern matching starts. Formally, $V^P_{\mathsf{SG}} = V^B_{\mathsf{SG}} \cup V^F_{\mathsf{SG}}$, and $V^B_{\mathsf{SG}} \cap V^F_{\mathsf{SG}} = \emptyset$.



(a) PROGRES / ReleaseRule (b) PROGRES / GiveRule

(c) FUJABA / ReleaseRule (d) FUJABA / GiveRule

Figure 4.4: Search plans of local search based tools

Definition 27 Given an adorned search graph ASG, a **search forest** E_{SF} is a subset of edges of the adorned search graph ASG such that for each free node exactly one of its incoming edges is selected as an edge of the search forest. Formally, $E_{\mathsf{SF}} \subseteq E_{\mathsf{SG}}$ such that

$$\forall \mathsf{v} \in V^F_{\mathsf{SG}}, \exists \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SF}}, \forall \mathsf{u}' \xrightarrow{\mathsf{z}'} \mathsf{v} \in E_{\mathsf{SG}} : \left(\mathsf{u}' \xrightarrow{\mathsf{z}'} \mathsf{v} \in E_{\mathsf{SF}} \iff (\mathsf{u} = \mathsf{u}' \wedge \mathsf{z} = \mathsf{z}')\right).$$

As a consequence, the search forest has as many edges as free nodes are contained by the adorned search graph. Formally, $|E_{\mathsf{SF}}| = \left|V^F_{\mathsf{SG}}\right|$.

Definition 28 Given a search forest SF in an adorned search graph ASG, a **search plan** $\mathsf{SP} : V_{\mathsf{ASG}} \to \mathbb{N}_{\left|V^F_{\mathsf{SG}}\right|}$ assigns non-negative integers up to the number of free nodes $\left|V^F_{\mathsf{SG}}\right|$ to nodes of adorned search graph ASG such that:

- The dummy node is labelled by 0. Formally, $\mathsf{SP}(\mathsf{d}) = 0$.
- Bound nodes are labelled by 0. Formally, $\forall \mathsf{x} \in V^B_{\mathsf{SG}} : \mathsf{SP}(\mathsf{x}) = 0$.
- Free nodes are labelled by positive integers. Formally, $\forall \mathsf{x} \in V^F_{\mathsf{SG}} : \mathsf{SP}(\mathsf{x}) > 0$.

- NAC nodes are labelled by non-negative integers. Formally, $\forall \mathsf{n}_i \in V_{\mathsf{SG}}^{\mathsf{NAC}} : \mathsf{SP}(\mathsf{n}_i) \geq 0$.
- Each positive label is attached to exactly one free node. Formally, $\forall j \in \mathbb{N} : 0 < j \leq \left|V_{\mathsf{SG}}^{F}\right| \Longrightarrow \big(\exists \mathsf{x} \in V_{\mathsf{SG}}^{F} : \mathsf{SP}(\mathsf{x}) = j \wedge \big(\forall \mathsf{y} \in V_{\mathsf{SG}}^{F} : \mathsf{SP}(\mathsf{x}) = \mathsf{SP}(\mathsf{y}) \Longrightarrow \mathsf{x} = \mathsf{y}\big)\big)$.
- Labels should always increase along search forest edges E_{SF}. Formally, $\forall \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SF}} : \mathsf{SP}(\mathsf{u}) < \mathsf{SP}(\mathsf{v})$.
- Labels cannot decrease along NAC check edges. Formally, $\forall \mathsf{v} \in V_{\mathsf{SG}}^{\mathsf{NAC}}, \forall \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SG}} : \mathsf{SP}(\mathsf{u}) \leq \mathsf{SP}(\mathsf{v})$.

4.2.4 A search plan description for constraint satisfaction based algorithms

Both constraint satisfaction and local search based strategies can be characterized by examining the order in which objects are assigned to unbound pattern nodes during the pattern matching phase. Now we present a notation for capturing the binding order uniformly for both kinds of strategies.

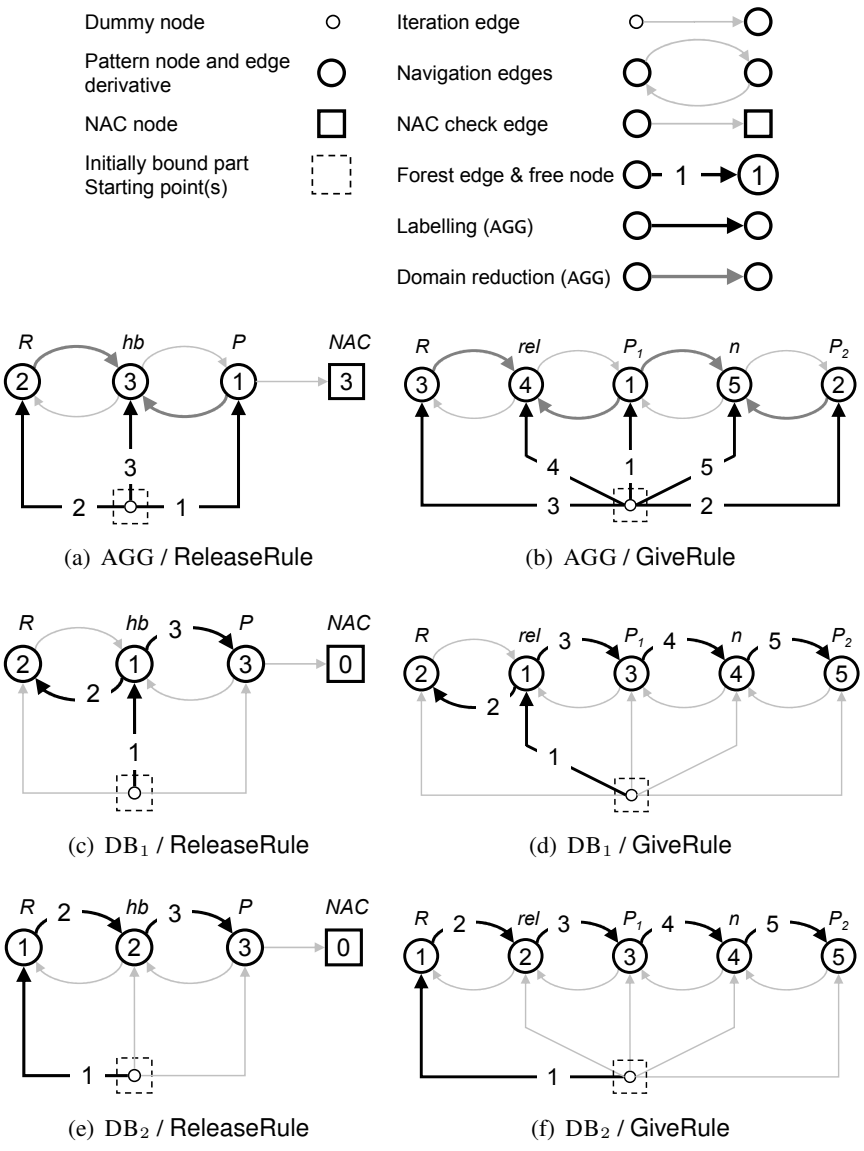
Algorithms that handle pattern matching as a constraint satisfaction problem do not directly involve the concept of search plans. However, the underlying constraint solver engine also has to define a variable binding order, which can be considered as a search plan derived dynamically at run-time. Now the major differences between the original search plan interpretation and its generalized version are discussed.

Both AGG and the DB approach unlike PROGRES and FUJABA treat links as autonomous entities having their own identifiers. As a consequence, edges of LHS and NAC patterns should also be considered as variables to which links should be assigned during pattern matching. In order to represent a variable binding order by a generalized search plan, not only nodes but also the edges of the precondition pattern have to be mapped to pattern node derivatives. This extension yields obvious modifications in the interpretation of iteration and navigation edges. In case of iteration edges, it means that iteration over all *links* of a given type is allowed (as in Fig. 4.5(d)). A generalized navigation edge can be interpreted as a navigation possibility between two pattern node derivatives of which one represents the pattern edge itself while the other denotes the adjacent source or target node of the edge.

AGG. By printing it into a text file, AGG directly supports the retrieval of the variable binding order being generated for the LHS of rules (but not for the NAC). The generalized search plan can be immediately constructed by using the retrieved binding order for the sequencing of pattern nodes.

Search plans being generated by AGG for the LHS of ReleaseRule and GiveRule are presented in Figs. 4.5(a) and 4.5(b), respectively. These search plans (with only black nodes and edges) would describe the worst case traversal of the search space tree in case of labelling without constraint propagation. As constraints typically express the adjacency of pattern edges to their source and target nodes, the propagation of such constraints would have an effect that is equivalent to the selection of the dark grey navigation edges (as in Figs. 4.5(a) and 4.5(b)). Since constraints are evaluated at run-time, it is impossible to derive a single static search plan, but it can be stated that concrete traversals would be some combinations of activities expressed by labelling edges selected from iteration edges and domain reduction edges selected from navigation edges.

DB approach. In case of the DB approach, the generalized search plans have been determined by manually analyzing the corresponding query plan provided by the query optimizer of the database. Search plans for preconditions of ReleaseRule and GiveRule are depicted in Figs. 4.5(c) and 4.5(e) and in Figs. 4.5(d) and 4.5(f), respectively.



(a) AGG / ReleaseRule (b) AGG / GiveRule

(c) DB_1 / ReleaseRule (d) DB_1 / GiveRule

(e) DB_2 / ReleaseRule (f) DB_2 / GiveRule

Figure 4.5: Tool-specific search plan interpretations

The DB approach is able to generate several static search plans (i.e., DB_1 and DB_2) for the same pattern depending on the actual content of the database (i.e., the model under transformation), and it constructs search plans separately for the LHS and NAC(s).

Search plans labelled with DB_1 have been generated for the different rules, when the model is the one presented in Fig. 3.4(b). By analyzing Fig. 4.5(d), it is worth examining release edges first as the model has no such edges, consequently, the pattern matcher can immediately determine that no matchings exist.

Search plans labelled with DB_2 have been produced for those test models that did not contain resources. In such cases, search plans, which start by testing resources first, detect pattern matching failure earlier.

Search plans of Figures 4.5(c) and 4.5(e) can check negative application conditions at first (as shown by the zeros in the NAC nodes) as matchings of the NAC pattern are available in a separate view (as going to be presented in Sec. 6.4.2), and the emptiness of this view can immediately indicate that each matching calculated for the LHS will be a matching for the whole rule as well.

4.2.5 Operations in search plan driven graph pattern matching

By definition, a search plan specifies the order of pattern nodes, in which they are mapped to objects during pattern matching. This order is directly denoted by the integers assigned to free nodes. Additionally, a search plan may also define elementary operations and order them to completely and precisely characterize the process of pattern matching. The search plan concept of this thesis supports this behaviour as well by means of defining mapping candidate generation tasks ($P(k,m)$) and check operations ($check(k,m')$) in Lines 4 and 7 of Algorithm 4.1, respectively.

More specifically, mapping candidate generation tasks are defined by edges of the search forest, while check operations are designated by such sets of search graph edges that have a common origin and that remain unselected by the search forest.

Mapping candidate generation is controlled by the following three rules:

- **Iteration.** If the source node d of the search forest edge leading to the free node v with label k is the dummy node, then pattern node $b(\mathsf{v})$ is scheduled to be mapped on the kth level of recursion. Objects that conform to the type of pattern node $b(\mathsf{v})$ are iterated as candidates for the mapping of pattern node $b(\mathsf{v})$. Formally,

$$P(k,m) = \left\{ (b(\mathsf{v}),c) \mid \mathsf{d} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SF}} \wedge \mathsf{SP}(\mathsf{v}) = k \wedge c \in V_M \wedge t(b(\mathsf{v})) \overset{*}{\leftarrow} t(c) \right\}$$

- **Navigation in forward direction.** If the search forest edge $\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v}$ connects pattern node derivative u to free node v with label k and goes in the *same* direction as its origin search graph edge $u \xrightarrow{z} v$ then pattern node v is scheduled to be mapped on the kth level of recursion. Possible candidates for the mapping of pattern node v are such objects, which conform to the type $t(v)$ of the target pattern node v, and which can be reached from the object c mapped to source pattern node u by navigating along edges of type $t(z)$. Formally,

$$P(k,m) = \Big\{ (b(\mathsf{v}),d) \mid \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SF}} \wedge \mathsf{SP}(\mathsf{v}) = k \wedge \mathsf{u} \in V^P_{\mathsf{SG}} \wedge b(\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v}) = u \xrightarrow{z} v \wedge$$
$$\exists c \xrightarrow{e} d \in E_M \,:\, m(u) = c \wedge t(v) \overset{*}{\leftarrow} t(d) \wedge t(z) = t(e) \Big\}$$

- **Navigation in backward direction.** If the search forest edge $\mathsf{v} \overset{\mathsf{z}_{inv}}{\rightarrow} \mathsf{u}$ connects pattern node derivative v to free node u with label k and goes in the *opposite* direction as its origin search graph edge $u \overset{z}{\rightarrow} v$ then pattern node u is scheduled to be mapped on the kth level of recursion. Possible candidates for the mapping of pattern node u are such objects, which conform to the type $t(u)$ of the source pattern node u, and which can be reached from the object d being mapped to target pattern node v by navigating along edges of type $t(z)$ in reverse direction. Formally,

$$P(k,m) = \Big\{ (b(\mathsf{u}), c) \mid \mathsf{v} \overset{\mathsf{z}_{inv}}{\rightarrow} \mathsf{u} \in E_{\mathsf{SF}} \wedge \mathsf{SP}(\mathsf{u}) = k \wedge \mathsf{v} \in V^P_{\mathsf{SG}} \wedge b(\mathsf{v} \overset{\mathsf{z}_{inv}}{\rightarrow} \mathsf{u}) = u \overset{z}{\rightarrow} v \wedge \\ \exists c \overset{e}{\rightarrow} d \in E_M \,:\, m(v) = d \wedge t(u) \overset{*}{\leftarrow} t(c) \wedge t(z) = t(e) \Big\}$$

The main check operation being executed on the kth level of recursion can be considered as the conjunction (logical AND) of simple operations, each of which is either an edge existence validation or a negative application condition check. Formally,

$$check(k,m) = check_{edge}(k,m) \wedge check_{\mathsf{NAC}}(k,m)$$

The two types of simple operations are as follows.

- **Checking edge existence.** If pattern node derivative v has label k, which is, in turn, at least as large as the label of pattern node derivative u, and these derivatives are bidirectionally connected by such edges that have a common origin pattern edge $u \overset{z}{\rightarrow} v$, and that are not in search forest SF, then this pattern edge $u \overset{z}{\rightarrow} v$ is scheduled to be checked on the kth level of recursion. This operation prescribes the existence of a link $c \overset{e}{\rightarrow} d$ connecting object c to object d, which can be mapped by the checked pattern edge $u \overset{z}{\rightarrow} v$ in such a way that this mapping preserves the type conformance of edges, source and target nodes. Formally,

$$check^{\mathsf{v}}(k,m) = \forall \mathsf{u}, \mathsf{v} \in V^P_{\mathsf{SG}},\, \forall \mathsf{u} \overset{\mathsf{z}}{\rightarrow} \mathsf{v} \in E_{\mathsf{SG}} \backslash E_{\mathsf{SF}},\, \forall \mathsf{v} \overset{\mathsf{z}_{inv}}{\rightarrow} \mathsf{u} \in E_{\mathsf{SG}} \backslash E_{\mathsf{SF}},\, \forall u \overset{z}{\rightarrow} v \in E_{\mathsf{LHS}} : \\ (\mathsf{SP}(\mathsf{u}) \leq \mathsf{SP}(\mathsf{v})) \wedge (\mathsf{SP}(\mathsf{v}) = k) \wedge \Big(b(\mathsf{u} \overset{\mathsf{z}}{\rightarrow} \mathsf{v}) = b(\mathsf{v} \overset{\mathsf{z}_{inv}}{\rightarrow} \mathsf{u}) = u \overset{z}{\rightarrow} v \Big) \Longrightarrow \\ \exists c \overset{e}{\rightarrow} d \in E_M \,:\, m(u) = c \wedge t(u) \overset{*}{\leftarrow} t(c) \wedge m(v) = d \wedge t(v) \overset{*}{\leftarrow} t(d) \wedge t(z) = t(e)$$

  A similar formula can be written for the case, when pattern node derivative u has the possibly larger label.

$$check^{\mathsf{u}}(k,m) = \forall \mathsf{u}, \mathsf{v} \in V^P_{\mathsf{SG}},\, \forall \mathsf{u} \overset{\mathsf{z}}{\rightarrow} \mathsf{v} \in E_{\mathsf{SG}} \backslash E_{\mathsf{SF}},\, \forall \mathsf{v} \overset{\mathsf{z}_{inv}}{\rightarrow} \mathsf{u} \in E_{\mathsf{SG}} \backslash E_{\mathsf{SF}},\, \forall u \overset{z}{\rightarrow} v \in E_{\mathsf{LHS}} : \\ (\mathsf{SP}(\mathsf{v}) \leq \mathsf{SP}(\mathsf{u})) \wedge (\mathsf{SP}(\mathsf{u}) = k) \wedge \Big(b(\mathsf{u} \overset{\mathsf{z}}{\rightarrow} \mathsf{v}) = b(\mathsf{v} \overset{\mathsf{z}_{inv}}{\rightarrow} \mathsf{u}) = u \overset{z}{\rightarrow} v \Big) \Longrightarrow \\ \exists c \overset{e}{\rightarrow} d \in E_M \,:\, m(u) = c \wedge t(u) \overset{*}{\leftarrow} t(c) \wedge m(v) = d \wedge t(v) \overset{*}{\leftarrow} t(d) \wedge t(z) = t(e)$$

  The overall formula for edge existence checks is the following

$$check_{edge}(k,m) = \bigwedge check^{\mathsf{u}}(k,m) \wedge \bigwedge check^{\mathsf{v}}(k,m).$$

- **Checking negative application condition.** For each NAC node with label k, a corresponding NAC checking operation is scheduled on the kth level of recursion, which prescribes that the execution of the full-featured pattern matching algorithm `match` (see Algorithm 4.1) for the

NAC pattern $b(\mathsf{v})$ (denoted temporarily by $\mathtt{match}_{\mathsf{NAC}}$) being invoked with any initial matchings m' that map derivatives u of shared nodes $b(\mathsf{u})$ to the same objects as matching m should return an empty set as a result. Formally,

$$check_{\mathsf{NAC}}(k,m) = \forall \mathsf{v} \in V_{\mathsf{SG}}^{\mathsf{NAC}} : \mathsf{SP}(\mathsf{v}) = k \Longrightarrow$$
$$\forall m' : V_{b(\mathsf{v})} \to V_M, \forall \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SG}} : m(b(\mathsf{u})) = m'(b(\mathsf{u})) \Longrightarrow \mathtt{match}_{\mathsf{NAC}}(1, m') = \emptyset.$$

4.2.6 Implementing a search plan driven pattern matcher

Note that Algorithm 4.1 has been presented as a recursive procedure only for presentation purposes. For efficiency reasons, implementations of pattern matching engines typically use the iterative equivalents of Algorithm 4.1. In case of compiled approaches, the procedure is represented as iterations embedded into each other. Levels of iterations show a one-to-one correspondence to the levels of recursion in such a way that the first level corresponds to the outermost loop, while the deepest level of recursion is represented by the innermost loop.

Example 13 Listings 4.1 and 4.2 present typical program codes for the search plans of Figs. 4.4(b) and 4.4(d), respectively.

```
1  // Level 1 -- Binds r : Resource
2  Iterator<Resource> iR = getAllResources();
3  while (iR.hasNext()) {
4    Resource r = iR.next();
5
6    // Level 2 -- Binds p1 : Process
7    Iterator<Process> iP1 = r.getReleaseTrg();
8    while (iP1.hasNext()) {
9      Process p1 = iP1.next();
10
11     // Level 3 -- Binds p2 : Process
12     Iterator<Process> iP2 = p1.getNextTrg();
13     while (iP2.hasNext()) {
14       Process p2 = iP2.next();
15       // p1, p2, and r now
16       // constitute a matching
17     }
18   }
19 }
```

Listing 4.1: Java program code equivalent of Fig. 4.4(b)

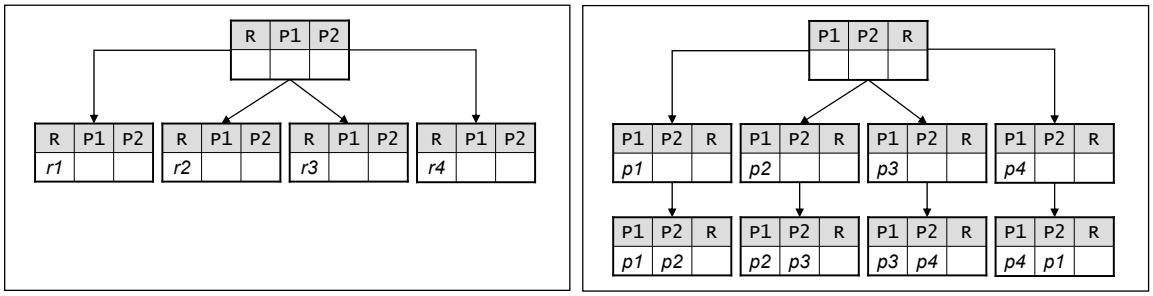
When the search plan of Fig. 4.4(b) is executed (by Listing 4.1), the pattern matching engine first tries to find an appropriate Resource for the pattern node R by querying all the Resources in the model. By supposing that the current model is the one presented in Fig. 3.4(b), the engine enumerates resources r1, r2, r3, and r4 in the outermost loop.

When pattern node R has been bound to resource r1, the current partial matching (i.e., which consists of the binding R to r1) is attempted to be extended by navigating along all release edges leading out of the already bound resource r1. As no such edges exist, the second `while` loop is not executed.

Due to the iterative nature of the matching of node R, all other resources are also tested. As no resources have outgoing release edges, each iteration ends with failure at the head of the second `while` loop.

The traversed search space tree while executing Listing 4.1 is depicted in Fig. 4.6(a). The root of this SST corresponds to the initial empty partial matching. The second level contains those snapshots that represent matchings produced by the outermost loop. As the second `while` loop has never been executed, there are no snapshots on the third level.

As it can be seen in Fig. 4.6(a), the pattern matching engine using the search plan of Fig. 4.4(b) tests 5 matching combinations altogether.



(a) SST1 (b) SST2

Figure 4.6: Search space trees traversed while executing Listings 4.1 and 4.2

When the search plan of Fig. 4.4(d) is executed, matchings for pattern nodes are searched in the P1, P2, R order as shown by Listing 4.2. The execution of this search plan produces 9 partial matching combinations altogether (as shown by Fig. 4.6(b)) as it starts with an empty partial matching, then pattern node P1 can be set to processes p1, p2, p3, and p4, which results in 4 independent combinations. Additionally, each partial matching can be extended by mapping pattern node P2 to exactly one process, which results in further 4 partial matchings.

```
1  // Level 1 -- Binds p1 : Process
2  Iterator<Process> iP1 = getAllProcesses();
3  while (iP1.hasNext()) {
4    Process p1 = iP1.next();
5
6    // Level 2 -- Binds p2 : Process
7    Iterator<Process> iP2 = p1.getNextTrg();
8    while (iP2.hasNext()) {
9      Process p2 = iP2.next();
10
11     // Level 3 -- Binds r : Resource
12     Iterator<Resource> iR = p1.getReleaseSrc();
13     while (iR.hasNext()) {
14       Resource r = iR.next();
15       // p1, p2, and r now
16       // constitute a matching
17     }
18   }
19 }
```

Listing 4.2: Java program code equivalent of Fig. 4.4(d)

As a consequence, the search plan of Fig. 4.4(b) is expected to recognize earlier the impossibility

of matching GiveRule on the model of Fig. 3.4(b).

4.2.7 General approximation for the size of the search space tree

In order to provide a general approximation for the time complexity of a given search plan (i.e., the size of the state space being traversed during pattern matching), we may apply the cost estimation of Sec. 7.3.1. If p, r, n and l denote the number of Processes, Resources, next edges and release edges in the model, respectively, then the search plan of Fig. 4.4(b) expectedly hits $r + r\frac{l}{r} + (r\frac{l}{r})\frac{n}{p}$ matching combinations as (i) there are r different choices for fixing pattern node R, (ii) on average $\frac{l}{r}$ release edges lead out of each resource resulting in $r\frac{l}{r}$ for the approximation on the number of possible partial matchings, which already bind pattern nodes R and P1, and (iii) for each such partial matching there are on average $\frac{n}{p}$ extension possibilities by navigating along next edges. As a result of a similar calculation process, the search plan of Fig. 4.4(d) expectedly traverses $p + p\frac{n}{p} + (p\frac{n}{p})\frac{l}{p}$ combinations.

The above formulae can be simplified to $r+l+\frac{ln}{p}$ and $p+n+\frac{ln}{p}$, respectively. By comparing these expressions, it can be determined which search plan is better for a model, which contains p processes, r resources, n next edges, and l release edges.

4.3 Conclusion

In this chapter, a general purpose, graph pattern matching algorithm has been presented first, in which mapping candidate calculation and matching check tasks could be customized providing a framework for analyzing both the theoretical and practical complexity of all existing and upcoming heuristics. Then, the technique of search plan driven pattern matching has been discussed by defining the concepts of search graphs and search plans, by extending its heuristic description facilities towards constraint satisfaction based algorithms, by formally specifying the exact pattern matching process, and by presenting a Java implementation for the technique.

CHAPTER

5 Benchmarking Framework for Graph Transformation

In this chapter, I present a benchmarking framework for graph transformation tools to be able to quantitatively assess, compare and analyze the run-time performance of (i) already existing approaches, (ii) their optimization strategies, and (iii) also all the techniques and algorithms that are going to be presented in later chapters.

5.1 Motivation for benchmarking

The aim of benchmarking is to systematically measure the performance of a system under different and precisely defined circumstances (i.e., by using several parameter combinations and data sets for these measurements). Such measurements help system engineers in decision making, i.e., when a choice has to be made between different alternatives by providing a proper assessment on the system characteristics. Although the graph transformation community has several specification examples for determining the expressiveness of approaches, it lacks *systematic benchmarks for measuring the performance of different tools*.

Related work

A good theoretical overview on software engineering benchmarks is provided by [123], which presents the preconditions and the main consequences of a successful benchmarking process by using a case study of the reverse engineering community.

There is a large variety of benchmarks and facilities supporting experiment design in different fields of computer engineering.

- **Artificial intelligence.** In case of agent-based systems, the Common Lisp Analytical Statistics Package (CLASP) [28] provides a formal, model-based methodology with powerful statistical techniques for justifying the decisions made during the design of the agent architecture.
- **Relational databases.** TPC-C [137] is a benchmark issued by the Transaction Processing Performance Council for measuring the performance of on-line transaction processing (OTLP) systems. Its current version comprises of a mix of five concurrent transactions and nine types of tables with a wide range of record and population sizes.

- **Rule-based expert systems.** [20] presents a collection of test suites that can be used for assessing the performance of rule-based expert systems. Among these benchmarks, Manners [74] describes a depth-first search solution to the problem of seating arrangement for guests at a dinner party, Waltz is a diagram labeling problem, which gives a 3-dimensional interpretation of 2-dimensional lines, the Aeronautical Route Planner searches for the lowest cost route between two points, and Weaver [68] is a VLSI channel and box routing algorithm.

Several specification examples (mappings such as UML-to-XMI in QVT [109], object-relational [109], UML-to-EJB [78], UML-to-XSD [24]) exist for graph transformation approaches, but their main goal is to demonstrate the *expressiveness* of the given approach, and they omit the performance aspects of graph transformation tools.

Objectives

Thus, I propose a systematic method for quantitative benchmarking in order to evaluate the performance of graph transformation tools. Typical features of the graph transformation paradigm and various optimization strategies exploited in different tools are identified and categorized. Moreover, the speed-up effects of these optimization strategies are measured and compared.

Since MOF metamodels and models are frequently formalized as graphs and graph transformation is a popular technique for capturing model transformations as indicated by a large variety of tools presented in Sec. 3.3, conclusions being drawn from experiments on GT tools are expected to be valid for model transformation tools as well. This statement is validated by [116], which proposes graph transformation to be used for defining the semantics of QVT. Further considerations about the applicability and the limitations of the benchmarking framework can be found in Section 5.6.

My objectives in the current chapter are the following.

- I define terms and concepts for graph transformation specific benchmarking in Section 5.2.1.
- I determine the most common features of graph transformation problems and tools in Sections 5.2.2 and 5.2.3. Based on tool-specific properties I identify various optimization strategies that are used in several graph transformation tools.
- I design benchmarks for different application scenarios for model transformations. In this sense, I propose a benchmark for simulating the dynamic behaviour of a system defined in a visual language in Section 5.3. In addition, I specify a benchmark for a model refactoring scenario in Section 5.4.
- In Section 5.5, measurements are executed on the benchmark of Sec. 5.3 by using carefully selected parameter settings and combinations of optimization strategies, and measurement results are presented. Then I shortly analyze the effects of optimization methods.
- Section 5.6 concludes this chapter with summarizing its theoretical and practical relevance.

5.2 Benchmark features

For presenting the tool related concepts of this chapter, we selected AGG (version 1.3.0), FUJABA (version 4.3), PROGRES (version 11), and the DB approach of Chapter 6 as representatives as already argued in Sec. 3.4.

Now the benchmarking framework is introduced by proposing first the terminology and the most common features of benchmarking for graph transformation systems.

The aim of benchmarking is to systematically measure the performance of a system under different, precisely defined and deterministic (reproducable) circumstances. The criterion of determinism has a strong impact on test set definition, since theoretically, both the next rule to be applied and the matching on which the rule is applied are nondeterministically selected. In order to avoid both kinds of nondeterminism, we define "checkpoints", where the instance model must be the same for all runs. Moreover, only an iterative execution of one rule is allowed between two checkpoints. Naturally, the end of the whole transformation sequence should also be a checkpoint.

5.2.1 Definitions of benchmarking

By a *scenario* we mean a broad application field where the paradigm of graph transformation is applicable. In [153] three scenarios are mentioned such as model analysis, model transformation, and simulation of visual languages with dynamic operational semantics. A scenario typically has some informal characteristics (e.g., "the structure of the system does not significantly change during the transformation process").

A *benchmark example* is a well-known problem serving as an incarnation of a scenario as it fulfills all the informal characteristics. For instance, the Mutex defined with its metamodel of Fig. 3.4(a) and graph transformation rules of Fig. 3.5 can be considered as a benchmark example for the simulation of visual languages as argued in Sec. 3.2. In technical terms, the metamodel and the graph transformation rules of the problem are fixed for a benchmark example, but instance models and concrete transformation sequences are left undefined.[1] In case of performance benchmarks, this decision can be justified by the requirement that prescribes a precise execution environment for the measurements. The investigation of qualitative characteristics such as learning curves, readability, reusability, etc., would also include the metamodel and the rules in the comparison.

A benchmark example may consist of several *test sets*. A test set is a complete, deterministic, but parametric specification. In this sense, the structure of both the instance model and the transformation sequence is fixed up to numerical parameters, which characterize, for instance, the model size, the length of the transformation sequence, etc. Moreover, we do not decide yet which optimization strategies for different tool features (see Sec. 5.2.3) are turned on/off in a test set.

In a *test case*, characteristics of the model and the transformation are still parametric, but we fix which optimization strategies (for details see 5.2.3) to turn on.

Finally, a test case is called a *run*, when even the runtime parameters are set. Thus, a run conforms to the requirements of determinism for benchmarking, since it is completely characterized by all its parameters and it is reproducable.

5.2.2 Paradigm features for graph transformation

A *paradigm feature* describes a characteristics of a problem. A *feature value* is a symbolic value corresponding to a numerical interval. Thus, each test set, test case and run is defined by representative feature values assigned to paradigm features. Note that it is difficult to determine crisp numbers for separating intervals of feature values in general, thus, we only define their typical order of magnitude. In case of graph transformation we identified the following paradigm features and feature values:

[1] To be precise, minor variations can be allowed in the metamodel within the same benchmark example due to practical reasons.

- *Pattern size*, or in other words the number of nodes and edges in the LHS graph, is a highly critical factor in the runtime behaviour of the pattern matching phase. As discussed in Sec. 4.1.2, the worst case complexity of graph pattern matching algorithms is exponential in the size of the pattern graph. On the other hand, in contrast to the size of patterns, RHS graph sizes have a linear influence on the runtime performance.

  Feature values: Since a benchmark problem may have several rules, the upper bound for the pattern sizes of all rules will be used as the value of the feature. Based on software engineering related considerations, large patterns typically consist of about 50 nodes and edges or more. Similarly, patterns with at most 10 nodes and edges are considered as small.

- The *maximum degree of nodes (fan-out)* in the model is the number of edges that are adjacent to a certain node. This feature has a significant impact on the complexity of a pattern matching algorithm which starts at a certain node and extends the match by examining its direct neighbourhood. To be more precise, only adjacent edges of the *same type* matter, since type checking typically precedes the enumeration of potential continuations during the pattern matching phase.

  Feature values: Values for this feature are also grouped into a small and a large category, which mean at most 30 and at least 100 outgoing edges, respectively. The latter limit is typically exceeded, if containment relations appear on the modeling level.

- The third feature of a test set is the *number of matchings*. In some cases it is sufficient to calculate only the first matching of a rule, but in other situations all the matchings have to be determined. It is obvious that in the latter case, this feature directly and seriously influences the overall runtime of the pattern matching. Note also that even in the first case, when a single matching needs to be calculated, the number of potential matchings to be checked can also significantly affect the performance.

  Feature values: The value of the feature is again the upper bound for the number of matchings in the pattern matching phases of all rule applications. The term small (large) is used, if at most 5 (at least 10) matchings exist.

- The *length of the transformation sequence* also affects the overall execution time. The more rule applications are performed, the longer it will take. However, this feature does not influence the average time needed for a single rule execution.

  Feature values: The value of this feature is the number of atomic rule executions performed. The term short (long) sequence is used, if the length is at most (more than) 1000.

Paradigm features, feature values, and the corresponding intervals are summarized in Table 5.1.

5.2.3 Tool features

Up to this point, features were completely dependent only on problem descriptions. Now we identify *tool features*, which are categories for typical optimization supported by different tools. For the moment four tool features are identified.

Tool feature 1: Parallel rule execution

In case of parallel rule execution, all matchings of a rule are calculated in the pattern matching phase, and then updates are performed as a transaction block on the collected matchings without re-evaluating

Paradigm feature	Feature value	Interval
Pattern size	small	≤ 10
	large	≥ 50
Fan-out	small	≤ 30
	large	≥ 100
Number of matchings	small	≤ 5
	large	≥ 10
Transformation sequence length	short	≤ 1000
	long	> 1000

Table 5.1: Paradigm feature summary

valid matchings during the transaction. For parallel rule executions we assume that the individual matchings are independent of each other.

Tool support and effects on search plans: Pseudo parallel (in other terms, concurrent) rule application is possible in all tools except for AGG,[2] and this technique does not modify the applied search plans. Pseudo parallel rule execution means that a transformation step could theoretically be performed on different matchings in parallel, but the implemented transformation engine sequentially applies the rule on the matchings due to architectural or other reasons. Note that none of the tools support real parallelism.

Tool feature 2: 'As long as possible' rule application

'As long as possible' (ALAP) rule application means an iterative execution of the selected rule. A standard graph rewriting step (with a pattern matching and an updating phase) is performed in each iteration as long as a matching can be found. A possible optimization strategy is to calculate independent matchings concurrently, and then to call the same procedure recursively.

The termination of the iteration should be guaranteed by the transformation designer. Thus, in order to avoid infinite loops, it must be ensured that the number of matching patterns always decreases, which is the simplest sufficient criterion for termination. For further termination criteria see [36, 87].

Tool support and effects on search plans: Since none of the examined tools supports ALAP rule application with optimized procedures, investigations on measuring the effects of this tool feature are omitted from this thesis.

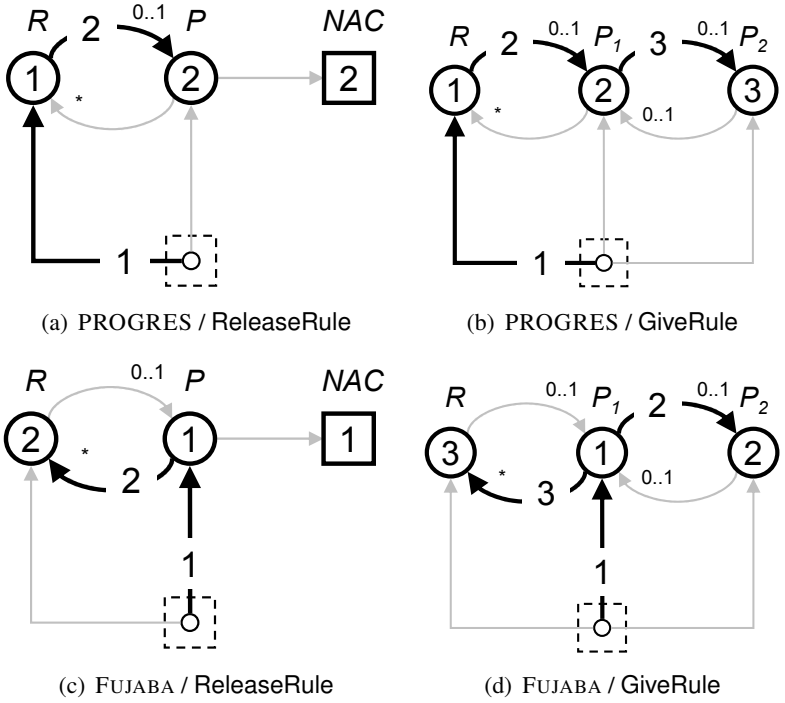
Tool feature 3: Multiplicity based optimization

Multiplicity based optimization is used, when a tool applies a different (and usually more powerful) strategy in order to find matching objects when navigating along a pattern edge with 0..1 multiplicity constraint on its target side. A typical strategy is to traverse 0..1 edges first in the pattern matching phase, since it yields a search tree that is narrower at the top-most levels.

Tool support and effects on search plans: PROGRES and FUJABA provide different methods for traversing edges with bounded multiplicity, while no such optimization strategies exist for AGG and the DB approach.

[2] AGG supports parallel rule execution since version 1.6.2 as an experimental feature.

Search plans describing the optimized strategies of PROGRES and FUJABA for ReleaseRule and GiveRule are presented in Fig. 5.1.



(a) PROGRES / ReleaseRule (b) PROGRES / GiveRule

(c) FUJABA / ReleaseRule (d) FUJABA / GiveRule

Figure 5.1: Effects of multiplicity based optimization strategies on search plans

Example 14 The program code representation of the unoptimized search plan of Fig. 4.4(b), which has already been presented in Listing 4.1 is repeated in Listing 5.1(a). The code equivalent of the search plan of Fig. 5.1(b) is presented in Listing 5.1(b). These code fragments are highly similar as the same order has been defined by the search plans of Fig. 4.4(b) and 5.1(b). However, due to multiplicity optimization, methods `getReleaseTrg()` and `getNextTrg()` now return a single object in Listing 5.1(b), which makes the innermost two `while` loops of Listing 5.1(a) unnecessary.

Tool feature 4: Parameter passing

Parameter passing provided between consecutive rule applications means that pattern matching in the subsequent rewriting steps is accelerated by directly reusing model elements passed as parameters without recalculating them in the later steps.

Tool support and effects on search plans: Parameter passing is supported by all four tools taking part in the measurements. Search plans, which have been generated by these tools for such versions of ReleaseRule and GiveRule that are able to handle parameters being passed, are presented in Fig. 5.2.

Search plans of Fig. 5.2 use exactly the same notation that have been introduced in Sec. 4.2.2 and 4.2.4. Note that the examined tools can handle different number of incoming parameters as it is expressed by the varying quantity of bound nodes.[3]

[3] FUJABA can handle more than one incoming parameters. However, in our implementation only a single parameter has been passed.

```
// Binds r : Resource
Iterator<Resource> iR =
    getAllResources();
while (iR.hasNext()) {
  Resource r = iR.next();

  // Binds p1 : Process
  Iterator<Process> iP1 = r.getReleaseTrg();
  while (iP1.hasNext()) {
    Process p1 = iP1.next();

    // Binds p2 : Process
    Iterator<Process> iP2 = p1.getNextTrg();
    while (iP2.hasNext()) {
      Process p2 = iP2.next();
      // p1, p2, and r now
      // constitute a matching
    }
  }
}
```

(a) Program code representation of Fig. 4.4(b)

```
// Binds r : Resource
Iterator<Resource> iR =
    getAllResources();
while (iR.hasNext()) {
  Resource r = iR.next();

  // Binds p1 : Process


  Process p1 = r.getReleaseTrg();

  // Binds p2 : Process


  Process p2 = p1.getNextTrg();
  // p1, p2, and r now
  // constitute a matching


}
```

(b) Program code representation of Fig. 5.1(b)

Listing 5.1: Effects of multiplicity based optimization strategies on the code generated for equivalent search plans

Example 15 Listing 5.2 presents the program code equivalent of the search plan of Fig. 5.2(a).

In this case, pattern nodes R and P have already been fixed, and their bindings are received by the pattern matching engine as input parameters in matching m. As a consequence, the engine simply has to check the non-existence of matchings for the negative application condition.

Table 5.2 summarizes the tool support for the presented optimization strategies.

Label + denotes a situation, when a strategy is supported by a given tool, and label – indicates the lack of support for a given optimization method. Since new heuristics can be discovered in the future, this set of tool features cannot be complete, but it already includes all widely supported representatives.

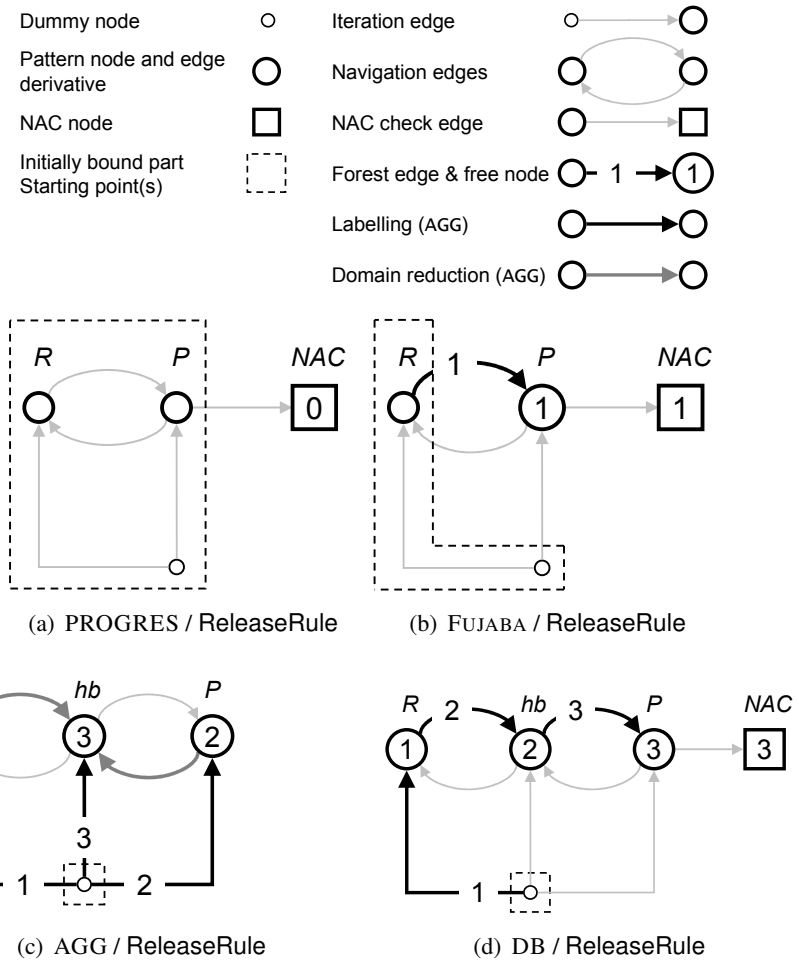
5.3 A benchmark example: Distributed mutual exclusion algorithm

I propose to select a distributed mutual exclusion algorithm as a benchmark example for the scenario of simulation of visual languages with dynamic operational semantics. This scenario can be characterized by a nearly static graph structure, where only a small part of the model is modified, and by short rewriting sequences that are executed many times during a simulation. Test sets are defined as rule application sequences that describe different possible runtime behaviours of the system.

The metamodel and the set of graph transformation rules needed to specify this benchmark example have already been introduced in Sections 3.2.1 and 3.2.2, respectively. For the mutual exclusion algorithm, three test sets have been defined.

5.3.1 The STS test set

This test set can be characterized by small LHS graphs. The length of transformation sequences, the number of fan-outs and the number of matchings are parameter dependent, so they are not distinguish-



(a) PROGRES / ReleaseRule (b) FUJABA / ReleaseRule

(c) AGG / ReleaseRule (d) DB / ReleaseRule

Figure 5.2: Effects of parameter passing on search plans

```
1  // Resource r and Process p are specified as inputs in Matching m
2  boolean match(Matching m) {
3    // Copies the value of the shared pattern node p to matching mNAC
4    Matching mNAC = new Matching();
5    mNAC.set(p, m.get(p));
6
7    // Checks NAC
8    if (! nacMatcher.match(mNAC)) {
9      // If no matchings exist for the NAC,
10     // then r and p constitute a matching for the precondition
11     return true;
12   }
13 }
```

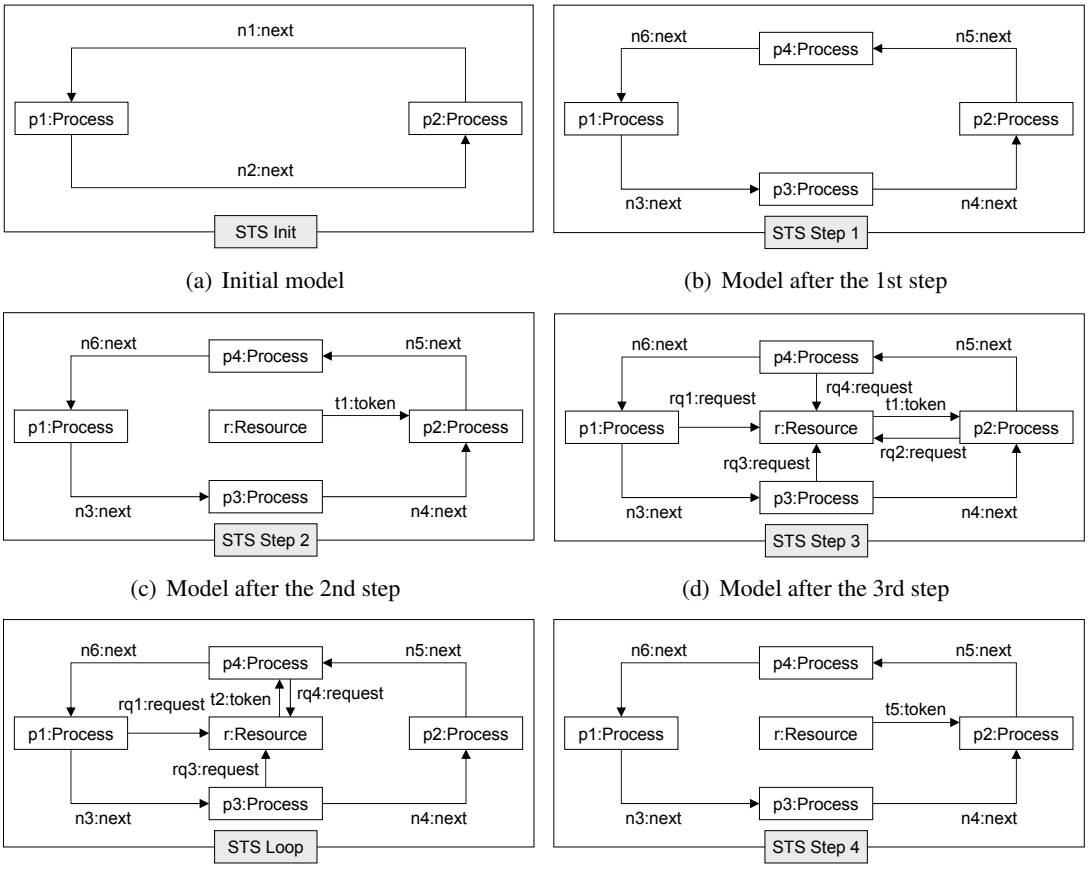
Listing 5.2: Program code equivalent of Fig. 5.2(a)

Tool features	AGG	PROGRES	FUJABA	DB
parameter passing	+	+	+	+
0..1 multiplicities	-	+	+	-
parallel execution	-	+	+	+
as long as possible	-	-	-	-

Table 5.2: Tool support for optimization strategies

ing features of this test set.

Initial instance models only contain two processes and two links of type next connecting the processes in both directions (as presented in Fig. 5.3(a)). The test set has one parameter N, which denotes the maximum number of processes appearing in the instance model during a specific run.



(a) Initial model

(b) Model after the 1st step

(c) Model after the 2nd step

(d) Model after the 3rd step

(e) Model after the first loop execution

(f) Model after the 4th step

Figure 5.3: Models in different phases of the STS test set when $N = 4$

The transformation sequence can be described as follows.

(a) NewRule (Fig. 3.5(a)) is applied first N–2 times in an arbitrary order. Since each application of NewRule adds a process to the token ring, after this step the instance model has a ring struc-

ture consisting of exactly N processes that are connected by N links of type next as shown in Fig. 5.3(b).

(b) The second step is to create a single resource by performing the MountRule (Fig. 3.5(c)) once. This rule also gives access rights to one of the processes, which is modeled by a newly created token link. The instance model is shown in Fig. 5.3(c).

(c) In the third step, each process issues a request for the single resource, which means the execution of RequestRule (Fig. 3.5(f)) for N times. Regardless of the execution order, the final instance model will be the one that is presented in Fig. 5.3(d). (So it is possible to apply RequestRule in parallel.)

(d) The final step handles the requests that have been issued in the previous step. To handle a single request TakeRule, ReleaseRule and GiveRule have to be applied in this specific order. In order to speed up pattern matching, parameter passing is possible among the rules that belong to the same loop.

TakeRule (Fig. 3.5(g)) assigns the process with the token to the resource by creating a held by link. Then ReleaseRule (Fig. 3.5(h)) lets the resource to be released by the process. Finally, the resource is released and the token is propagated to the next process in the token ring by the execution of GiveRule (Fig. 3.5(i)). The instance model at this point is shown in Fig. 5.3(e).

Since all the N processes have already requested the resource, the above-mentioned 3 rules have to be executed in a loop for N times, which results in a rule execution sequence of length $3N$. (Note that there exists only a single matching to which the subsequent rule can be applied at the time when the rule application is scheduled, so the rule execution order of the fourth step is fully deterministic.) In the end, the instance model will be the one that is depicted in Fig. 5.3(f).

The transformation sequence consists of $5N$–1 rule applications altogether. The largest instance model that appears during the rule application phase has N+1 objects and $2N$+1 links (see Fig. 5.3(d)).

Optimization possibilities.

- Instead of having zero-to-many multiplicities on all association ends, it is possible to restrict some of them to zero-to-one, as it is presented in the metamodel of Fig. 3.4(a). Since the model contains only a single resource, knowing and using this fact may cause performance improvements for some tools, since pattern matching can be started at this well-defined node.

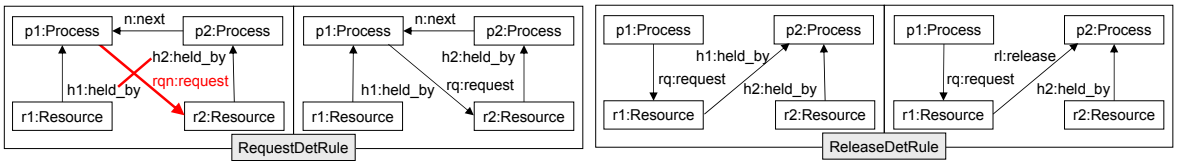
- As it was already mentioned in the test set description, the three rules in the loop of the fourth step may be applied in such a way that the selected processes and resources can be passed to consecutive rules as input parameters, which may speed-up pattern matching.

5.3.2 The LTS test set

This test set can be characterized by small LHS graphs, and small fan-outs. The number of matchings and the length of transformation sequences are parameter dependent, so they are not distinguishing features of this test set.

The LTS test set can be considered as such an extension of the STS test set, which uses all the rules defined by the benchmark example, and which has a separate parameter for setting the length of transformation sequences.

For this test set, two rules (namely, RequestRule and ReleaseRule) are modified in order to restrict their applicability in certain situations and to get a deterministic transformation sequence. The modified rules that ensure the deterministic request and release of resources are referred as RequestDetRule and ReleaseDetRule and they are depicted in Figures 5.4(a) and 5.4(b), respectively.



(a) RequestDetRule (b) ReleaseDetRule

Figure 5.4: Extra rules for the LTS test set

In this case, we have two runtime parameters (namely, N and R). N denotes the number of processes and resources in the initial instance model, and it influences both the model size and the length of the transformation sequence. We refer to a transformation sequence as a *basic execution unit*, if instance models before and after execution are isomorphic, and the sequence can be executed several times in a loop. The role of R is to determine how many times a basic execution unit is executed. As a consequence, R has influence only on the length of the transformation sequence.

The initial instance model consists of $2N$ objects (N processes and N resources) and $2N$ links. N links are of type next and they are used to organize processes into a token ring. The other N links mark processes holding resources in such a way that no held by links have common ends (i.e., each resource is held by at most one process and each process reserves at most one resource). A sample initial instance model is presented in Fig. 5.5(a) for the $N = 4$ case.

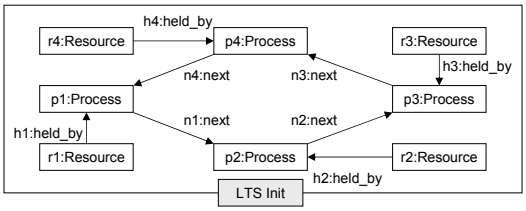
The transformation sequence inside the basic execution unit is defined as follows.

(a) As a first step, RequestDetRule (Fig. 5.4(a)) is applied N times. RequestDetRule selects two neighbouring processes each of which holding at least one resource, and the one that is ahead in the token ring, issues a request on the resource that is held by the other process, if it has not issued any requests yet on the same resource. The resulting instance model (see Fig. 5.5(b)) should be identical after any sequence of rule applications during the first step, so this set of rules can be applied in parallel.

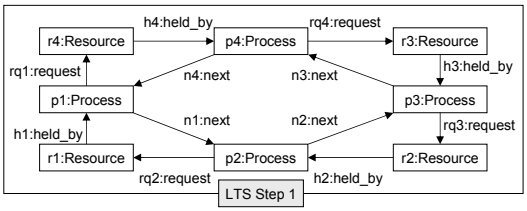
(b) This step is a single execution of BlockedRule (Fig. 3.5(j)), which initiates the deadlock detection algorithm by introducing a new blocked link. There are N matchings for this rule before its application, so the graph transformation engine can choose freely on which matching the selected rule is applied. The result of the rule application is something similar to Fig. 5.5(c).

(c) WaitingRule (Fig. 3.5(k)) is executed now N–1 times. Since the model contains only a single blocked link, this sequence is fully deterministic. Moreover, it describes how the blocked link is propagated in the token ring in the direction being marked by the set of next links. After this step, the blocked edge makes a whole round in the token ring as it is depicted in Fig. 5.5(d).

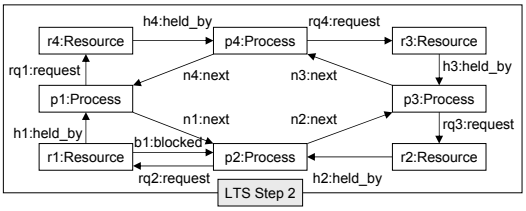
(d) Now a single execution of UnlockRule (Fig. 3.5(m)) follows, which can be done only on a single matching. This breaks the circular blocking situation that causes the deadlock by forcing a process to release its resource. The result will be a model that is shown in Fig. 5.5(e).



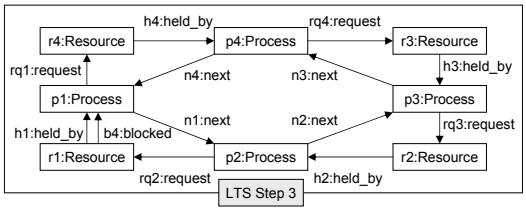
(a) Initial model with parameter $N = 4$



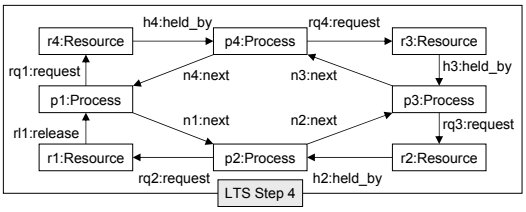
(b) Model after the 1st step



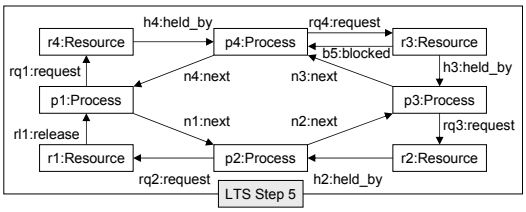
(c) Model after the 2nd step



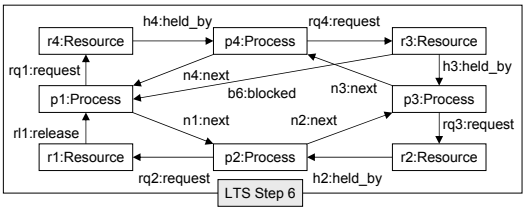
(d) Model after the 3rd step



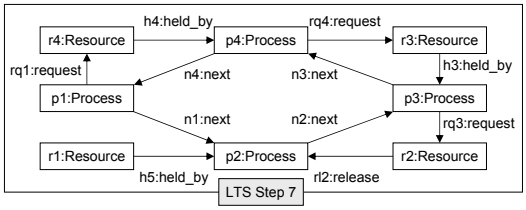
(e) Model after the 4th step



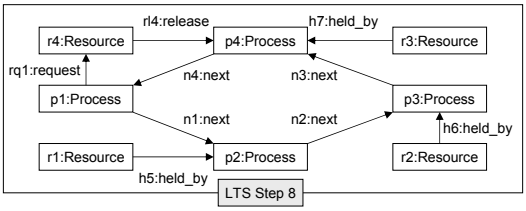
(f) Model after the 5th step



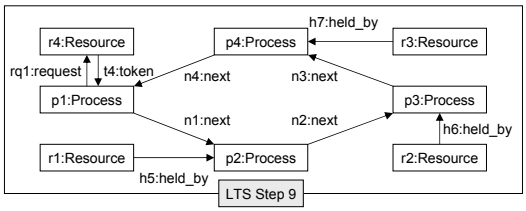
(g) Model after the 6th step



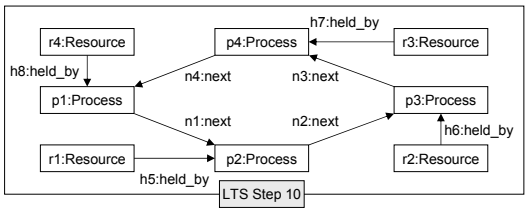
(h) Model after the 7th step



(i) Model after the 8th step



(j) Model after the 9th step



(k) Model after the 10th step

Figure 5.5: Models in different phases of long transformation sequence

(e) In the fifth step, BlockedRule (Fig. 3.5(j)) is executed once again, generating a new blocked link. In this case, the rule can be applied on N–1 possible matchings. Since this is a nondeterministic choice, the result will be something similar to Fig. 5.5(f).

(f) Now WaitingRule (Fig. 3.5(k)) is applied at most N–1 times. There exists only a single matching on which next rule application can be performed until the point, when the blocked edge points to the same process as the release edge (see Fig. 5.5(g)). From that point, no matchings can be found. The ratio of successful and unsuccessful rule application steps depends on the context on which the previous BlockedRule was executed.

(g) IgnoreRule (Fig. 3.5(l)) is executed once to restore the instance model we had after the fourth step (Fig. 5.5(e)) by deleting the blocked link.

(h) The eighth step is an execution of a loop that contains GiveRule, TakeRule and ReleaseDetRule in this specific order. The first execution of the loop yields the model of Fig. 5.5(h). In order to accelerate pattern matching, parts of matchings can be passed as parameters to the successive rule of the loop.

GiveRule (Fig. 3.5(i)) releases a resource that was held by a process, and gives the token to the next process in the ring. During the execution of TakeRule (Fig. 3.5(g)), the process that has a token for a requested resource, reserves it by introducing a held by edge between them. The ReleaseDetRule handles the release of a resource in a special context to ensure a deterministic execution order.

The loop is executed N–1 times altogether. Note that the cardinality of matchings of GiveRule is decreased by one after each loop execution. The resulting model we get after the eighth step is presented in Fig. 5.5(i).

(i) In the ninth step GiveRule is performed once on the single matching that still exists resulting in a model that is depicted in Fig. 5.5(j).

(j) The final step is a single TakeRule application again on the only possible matching, and the result (shown in Fig. 5.5(k)) will be isomorphic with Fig. 5.5(a). The single difference is that now each resource is held by the process that is one step ahead of the one that reserved the resource before the basic execution unit started.

A basic execution unit contains a transformation sequence of length $6N$+1. During the execution of such a basic unit the largest instance model has exactly $2N$ objects and at most $3N$+1 links as can be seen in Fig. 5.5(c). This unit was executed R times in our experiments resulting in the same upper bound for the model size and a transformation sequence of length of $R(6N$+1$)$.

5.3.3 The 'as long as possible' test set

The test set can be characterized by small LHS graphs, and small fan-outs. The length of transformation sequences and the number of matchings depend on the runtime parameter.

RequestRule has to be slightly modified again to ensure the appropriate behavior during the execution of this test set. The modified RequestRule will be referred to as RequestSimpleRule and it is depicted in Fig. 5.6.

This test set uses N as its single runtime parameter. N denotes the number of processes and resources in the system, and this parameter influences both the model size and the length of the transformation sequence.

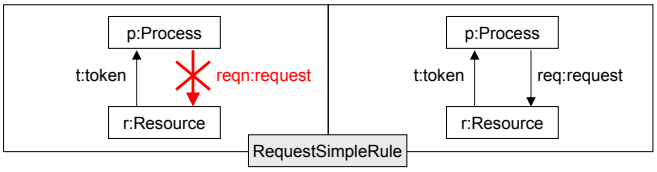
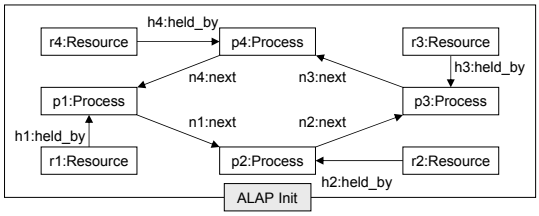
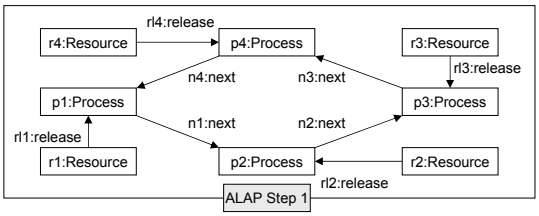


Figure 5.6: Simplified version of RequestRule

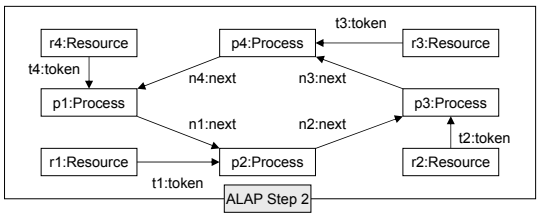
The initial instance model consists of $2N$ objects (N processes and N resources) and $2N$ links. Processes are arranged into a token ring along N links of type next. Furthermore, each resource is reserved by at most one process and each process holds at most one resource at a time. In the model, this property is expressed by N links of type held by. A sample initial instance model is presented in Fig. 5.7(a) for the case $N = 4$.



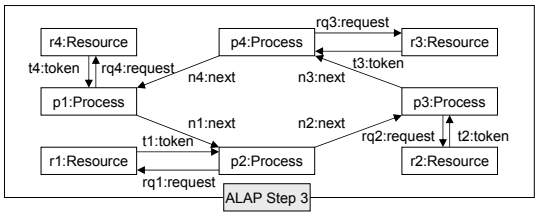
(a) Initial model with parameter $N = 4$



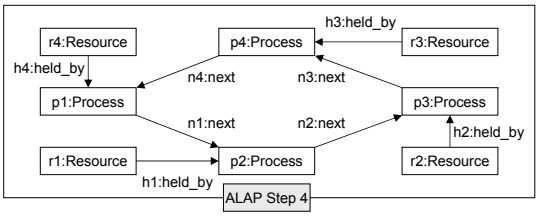
(b) Model after the 1st step



(c) Model after the 2nd step



(d) Model after the 3rd step



(e) Model after the 4th step

Figure 5.7: Models in different phases of 'as long as possible' rule execution

The transformation sequence of the test set consists of 4 macro steps. Each macro step is an iterative execution of a single rule.

(a) During the first step, ReleaseRule is executed N times, yielding a model (see Fig. 5.7(b)) where all the resources are now connected to their corresponding processes via a release link.

(b) Then the execution of GiveRule follows, which is performed again N times. This rule enables the next process in the ring to reserve the resource by giving the token to the process. The result model is depicted in Fig. 5.7(c).

(c) The iterative execution of RequestRule initiates a process to issue a request on the resource for which the process already has a token. As a result of an iteration of length N, we obtain the model of Fig. 5.7(d).

(d) Finally, TakeRule is executed N times. This rule assigns a process to a resource if the process has already reserved a token for the requested resource. The final instance model is isomorphic to the initial model. However, in the final model, a given resource is held by the next process in the token ring (see Fig. 5.7(e) vs. Fig. 5.7(a)).

Transformation sequence length and the number of matchings can be expressed as $4N$ and N, respectively.

Optimization possibilities.

- Since the order of rule applications inside a macro step is irrelevant, the specific rule can be applied concurrently (in parallel) on different processes. As a consequence, if each macro step consists of the parallel execution of the prescribed rule, then parallel and sequential transformations yield equivalent results.
- Moreover, each rule application of a macro step disables the execution of the same rule on the same process, it leaves the enabledness of the same rule on other processes unchanged, and finally, it enables the execution of the following rule on the same process. These observations yield an 'as long as possible' style application of rules appearing in the same macro step.

5.3.4 Feature matrix

A *feature matrix* summarizes the features of test sets. A sample feature matrix for the test sets of the mutual exclusion benchmark example is shown in Table 5.3. Rows of the upper and the lower table correspond to paradigm and tool features, respectively. Columns represent test sets. Moreover, these test sets can be grouped to form a benchmark example. Identifiers of the benchmark example (Mutex) and the test set (e.g., STS) are presented in the topmost two header fields of the column in turn. A field in the table contains the feature value that characterizes the given feature of a test set.

As the domain of feature values differ for paradigm and tool features, the possible values in the feature matrices are also different. Paradigm features may have values that have been defined in Sec. 5.2.2, or they may be parameter dependent (PD), which means that their categorization may vary depending on the runtime parameter settings. Tool features can be characterized by three values. Label ON (OFF) is used if the corresponding optimization strategy is applicable and it is switched on (off) in our measurements. Label N/A denotes that the optimization strategy for the tool feature is not applicable.

5.4 The object-relational mapping as a benchmark example

The object-relational mapping can be considered as an incarnation of a typical model transformation scenario, which can be characterized by a graph structure that always increases in size as the transformation progresses, and by rules with such negative application conditions that inhibit the repeated

Paradigm features	Mutex		
	STS	LTS	ALAP execution
LHS size (small/large)	small	small	small
fan-out (small/large)	PD	small	small
matchings (few/many)	PD	PD	PD
transformation sequence length (short/long)	PD	PD	PD

Tool features	Mutex		
	STS	LTS	ALAP execution
parameter passing	***ON/OFF***	OFF	N/A
0..1 multiplicities	***ON/OFF***	OFF	OFF
parallel execution	OFF	OFF	***ON/OFF***
as long as possible	N/A	OFF	OFF

Table 5.3: Paradigm and tool features of the mutual exclusion benchmark example

execution of the given rule on the same matching. The metamodel and the set of graph transformation rules needed to specify this benchmark example have already been introduced in Sections 2.2 and 3.1, respectively. For the object-relational mapping, only a single test set has been defined.

Test set specification

This test set can be characterized by patterns from the range between the small and the large categories. The number of matchings, the maximum degree of nodes (i.e., the fan-out) and the length of the transformation sequence are parameter dependent.

The single runtime parameter N denotes the number of Classes in the initial instance model, and it influences both the model size and the transformation sequence length on a quadratic scale.

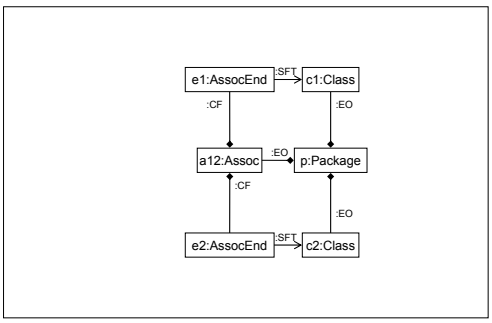
The initial instance model has a single Package that contains N Classes. An Association and two AssociationEnds are added to the model for each pair of Classes, thus initially, we have $N(N-1)/2$ Associations and $N(N-1)$ AssociationEnds. Associations are also contained by the single Package as expressed by the corresponding links of type EO. Each AssociationEnd is connected to a corresponding Association and Class by a CF and SFT link, respectively. A sample initial model is presented in Fig. 5.8(a) for the $N = 2$ case.

The transformation sequence consists of 4 macro steps that are executed in this specific order.

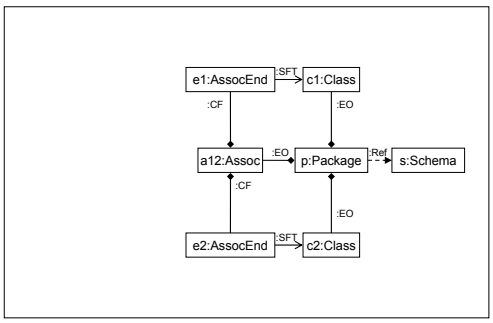
(a) The first macro step is a single application of PackageRule (Fig. 3.1(a)), which results in a model shown in Fig. 5.8(b).

(b) This is followed by a macro step that consists $N(N-1)/2$ applications of AssociationRule (Fig. 3.1(c)). The model at this point is depicted by Fig. 5.8(c).

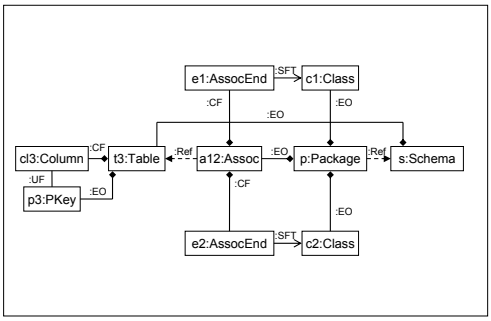
(c) Then classes are transformed by the execution of ClassRule (Fig. 3.1(b)) for N times. The resulting model after the intermediate step of the $N = 2$ case is presented in Fig. 5.8(d), while Fig. 5.8(e) shows the model obtained at the end of this macro step.



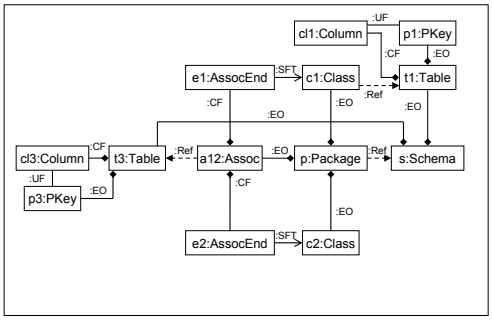
(a) Initial model for the $N = 2$ case



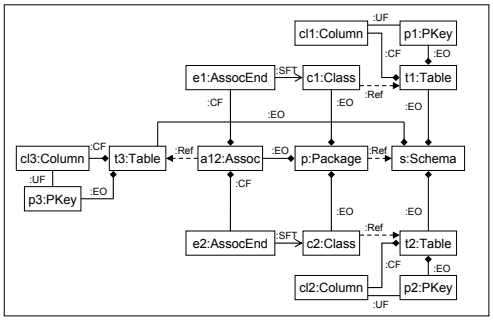
(b) Model after the 1st macro step



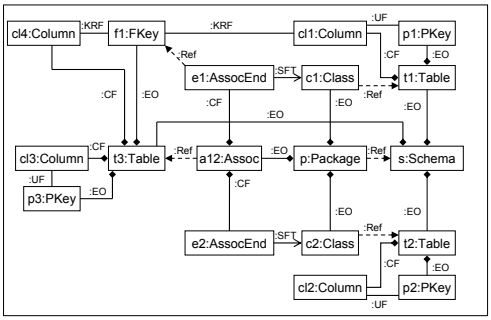
(c) Model after the 2nd macro step



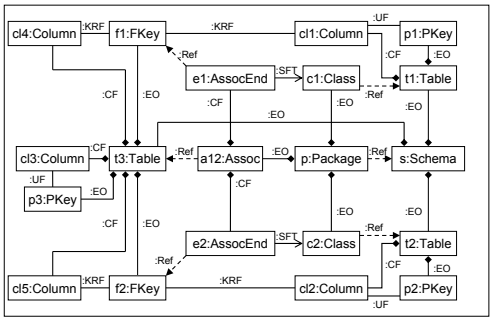
(d) Model under transformation during the 3rd macro step



(e) Model after the 3rd macro step



(f) Model under transformation during the 4th macro step



(g) Model after the 4th macro step

Figure 5.8: Models in different phases of rule execution

(d) Finally, a macro step of length $N(N-1)$ follows, which prescribes the application of AssocEndRule (Fig. 3.1(f)). Models resulted by the intermediate and final steps are depicted in Figures 5.8(f) and 5.8(g), respectively.

The transformation sequence consists of $(3N^2 - N + 2)/2$ rule applications altogether. The largest instance model has $4N^2 + 2$ objects and $(21N^2 - 7N + 4)/2$ links as shown by Fig. 5.8(g).

Optimization possibilities.

- Since the order of rule applications inside a macro step is irrelevant, the specific rule can be applied concurrently (in parallel). As a consequence, if each macro step consists of the parallel execution of the prescribed rule, then parallel and sequential transformations yield equivalent results.
- Since negative application conditions in the rule set of this benchmark example inhibit the repeated execution of the given rule on the same matching, an 'as long as possible' style rule application is allowed for each macro step.

5.5 Measurement results

In order to assess the acceleration effects caused by the different optimization strategies, measurements have been carried out on all test sets of the mutual exclusion benchmark example described in Sec 5.3. Test cases have been selected for these measurements according to the following principles.

(a) In order to obtain a feasible method, each optimization strategy is enabled only for the test set, where the effect of optimization is the most significant.

(b) Only selected combinations of switching on and off tool optimization strategies are carried out, since the analysis of all possible combinations would be practically infeasible as it requires unacceptably high effort even for a single test set.

By following the above guidelines, only 7 test cases have been selected for our measurements instead of the original 32 test cases (that correspond to all possible combinations of ONs and OFFs). Note that measurements for the ALAP style rule execution is omitted, since no optimization strategies are built into existing tools.

In order to assess the performance of graph transformation tools, tests were performed on a 1500 MHz Pentium machine with 768 MB RAM. A Linux kernel of version 2.6.7 served as the underlying operating system.

All the runs were executed without the GUI of tools, so rule applications were guided by Java programs (except for the measurements for PROGRES, where C programs were used). This way, we were doing programmed graph rewriting in each case for batch transformations.

Our general guideline for the execution of measurements was to use the standard services available in the default distribution of the tools, fine-tuned according to the suggestions of different tool developers. For instance, we exploited a parameter passing strategy of AGG, which is only available in programmed mode. In case of FUJABA, the models were slightly altered to provide better performance. We used GRAS as being the default underlying graph-oriented database for the PROGRES tests, and in addition, the Prolog-style cuts in the specification to make the execution deterministic. Moreover, the graphical user interface of PROGRES was switched off during the measurements as we prepared

the compiled version of the specification. In case of database tests, MySQL (version 4.1.7) with the default configuration was used as the underlying relational database using the built-in query optimization strategies.

	Mutex (STS)		Proc. #	Model size #	TS length #	AGG match msec	AGG update msec	PROGRES match msec	PROGRES update msec	Fujaba match msec	Fujaba update msec	DB match msec	DB update msec
GiveRule	multiplicity opt.	OFF	10	32	49	2.89	2.24	0.40	0.09	0.18	0.15	5.02	31.42
	param. passing	OFF	100	302	499	5.18	10.58	0.31	0.20	0.27	0.14	7.10	33.38
	parallel exec.	OFF	1000	3002	4999	-	-	0.63	0.26	0.35	0.03	4.26	32.13
	multiplicity opt.	ON	10	32	49			0.28	0.18	0.17	0.13		
	param. passing	OFF	100	302	499			0.14	0.09	0.19	0.15		
	parallel exec.	OFF	1000	3002	4999			0.49	0.28	0.08	0.03		
ReleaseRule	multiplicity opt.	OFF	10	32	49	3.16	1.54	0.26	0.06	0.19	0.22	18.99	48.38
	param. passing	OFF	100	302	499	3.12	9.11	11.71	0.18	0.70	0.17	12.87	55.86
	parallel exec.	OFF	1000	3002	4999	-	-	249.23	1.25	2.11	0.04	32.86	49.99
	multiplicity opt.	ON	10	32	49			0.20	0.16	0.19	0.22		
	param. passing	OFF	100	302	499			0.10	0.08	0.63	0.12		
	parallel exec.	OFF	1000	3002	4999			0.48	0.29	2.10	0.04		
	multiplicity opt.	OFF	10	32	49	2.65	1.87	0.16	0.08	0.14	0.23	7.32	51.48
	param. passing	ON	100	302	499	2.89	13.19	0.30	0.17	0.15	0.18	11.96	48.85
	parallel exec.	OFF	1000	3002	4999	-	-	0.49	0.40	0.03	0.03	-	-
	multiplicity opt.	ON	10	32	49			0.31	0.15	0.14	0.22		
	param. passing	ON	100	302	499			0.12	0.07	0.15	0.12		
	parallel exec.	OFF	1000	3002	4999			0.47	0.23	0.03	0.03		
	(LTS)												
Release	multiplicity opt.	OFF	4	21	2500	1.86	17.55	0.62	0.15	0.15	0.09	4.15	34.01
	param. passing	OFF											
	parallel exec.	OFF	1000	5001	60001	1116.34	871.32	269.58	0.62	0.26	0.03	20.47	29.35
	(ALAP)												
ReleaseRule	multiplicity opt.	OFF	10	50	40	19.37	5.93	0.34	0.08	2.21	0.18	7.38	33.56
	param. passing	OFF	100	500	400	7.02	7.57	20.08	0.37	0.56	0.17	8.81	50.52
	parallel exec.	OFF	1000	5000	4000	81.34	148.91	242.95	0.85	0.60	0.10	24.12	62.06
	multiplicity opt.	OFF	10	50	40			0.10	0.19	2.17	0.19	1.23	0.78
	param. passing	OFF	100	500	400			0.16	0.08	0.19	0.17	0.54	1.65
	parallel exec.	ON	1000	5000	4000			0.38	0.06	0.09	0.11	0.81	0.90

Table 5.4: Experimental results

Table 5.4 presents the average of execution times measured in 3 runs.

- The head of a row (i.e., the first two columns) shows the name of the rule and the optimization strategy configuration on which the average is calculated. (Note that a rule is executed several times in a run.)

- The third column (Proc) depicts the number of processes in the run, which is, in turn, the runtime parameter N for the test case.

- The fourth and fifth columns show the concrete values for the model size and the transformation sequence length, respectively.

- Values in match and update columns depict the average times needed for a single execution of a rule in the pattern matching and updating phase, respectively.

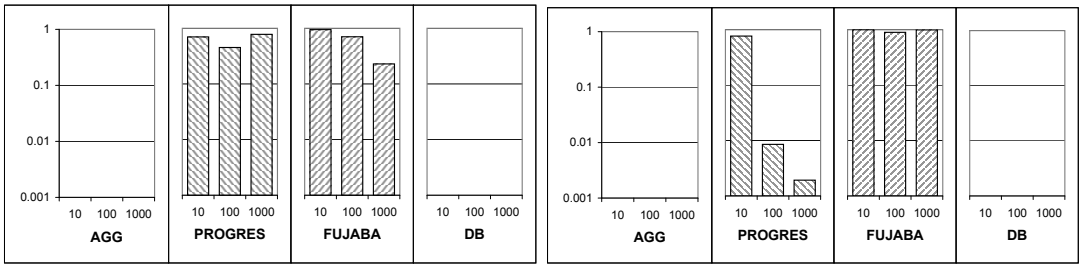
Execution times were measured on a microsecond scale, but a millisecond scale is used in Table 5.4 for presentation purposes. Light grey areas denote the lack of support for a combination of optimization strategies by a given tool.

By evaluating the bare measured values, it can be observed that the execution times needed for pattern matching and updating are frequently on the same order of magnitude (like e.g., in case of FUJABA).

- This may be caused by the fact that sophisticated tools can prepare good search plans for graph pattern matching in certain cases (even resulting in a linear execution time, which is comparable to the theoretical time complexity of the updating phase), while in other cases graph pattern matching is significantly slower. This observation, in turn, justifies that pattern matching heuristics are worth being examined and optimized.

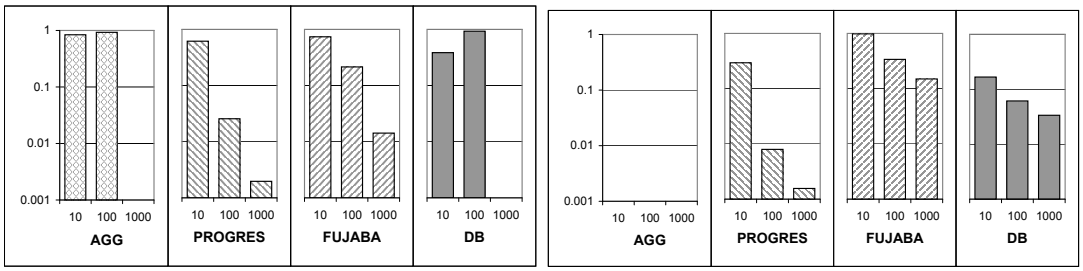
- In case of the DB approach, the transaction handling (and other consistency preservation) tasks, which are typically performed in the updating phase, may be another reason for the comparable values.

The further comparison of bare values could result in misleading consequences, due to the fact that the examined tools use significantly different techniques for pattern matching as discussed in Chapter 4. As a consequence, the remaining analysis focuses on the *speed-up effects caused by the different optimization strategies*, which can be perfectly presented by normalized charts as in Fig. 5.9.



(a) Effects of multiplicity based optimization for GiveRule

(b) Effects of multiplicity based optimization for ReleaseRule



(c) Effects of parameter passing

(d) Effects of parallel rule execution

Figure 5.9: Experimental results

Each subfigure of Fig. 5.9 shows the acceleration effect caused by a single optimization strategy being switched on. Mainly the ReleaseRule has been used to present the measurement results as in Figures 5.9(b), 5.9(c), and 5.9(d) for the evaluation of multiplicity based optimization, parameter passing and parallel rule execution, respectively. However, effects of multiplicity based optimization are also examined on GiveRule as depicted in Fig. 5.9(a).

A block in a subfigure consists of similarly decorated columns, and it represents measurement results belonging to the same tool, whose name is also indicated under the block itself. Each column corresponds to a run with a fixed run-time parameter setting being shown on the horizontal axis. Recall that model size and transformation sequence length are linear functions of the parameter in our benchmark example.

For each tool and run-time parameter setting, the height of the corresponding column denotes the ratio of the average execution times in optimized and unoptimized solutions required for the pattern matching phase of a single application of the rule in turn. Note that reference levels with value 1 correspond to tool and run specific absolute execution time values being measured, which are typically different for each column. Ratios are presented on a logarithmic scale. Missing columns in a whole block denote the lack of support for an optimization strategy in a given tool. However, missing columns for AGG and DB at model size 1000 in Fig. 5.9(c) show that the runtime limit has been exceeded in those cases.

From the measurements of Figure 5.9, we can make the following observations.

Effects of multiplicity based optimization. The effect of multiplicity based optimization is undetectable, if optimized and unoptimized versions of search plans are exactly the same. Such a situation can be experienced in case of FUJABA (see the FUJABA block in Fig. 5.9(b)), when search plans being created for the ReleaseRule are analyzed by comparing Figs. 5.1(c) and 4.4(c).

If strategies only differ in the applied navigation methods (see the small 0..1 on navigation edge connecting P_1 to P_2 as the only difference between Figures 5.1(d) and 4.4(d)), then a moderately growing trend of speed-up can be experienced as depicted in the FUJABA block of Fig. 5.9(a).

If the optimized and unoptimized search plans have structural differences as in case of PROGRES for the ReleaseRule (being depicted in Figs. 5.1(a) and 4.4(a), respectively), then a heavily decreasing tendency in average pattern matching time can be observed as in the PROGRES block of Fig. 5.9(b).

For the multiplicity based optimization techniques, as a general conclusion we may state that the speed-up of pattern matching can be derived mostly from the restructuring of search plans.

Effects of parameter passing. The gain from parameter passing as shown in Fig. 5.9(c) is noticable for FUJABA and PROGRES, while AGG and DB approaches cannot benefit from this tool feature. In case of AGG, parameter passing is not officially supported, i.e., it was programmed manually for the measurements. In the DB case, optimizations for parameter passing are carried out automatically by the query optimizer of the database.

By analyzing search plans with parameter passing support (such as Fig. 5.2(a) for PROGRES, Fig. 5.2(b) for FUJABA, Fig. 5.2(c) for AGG, and Fig. 5.2(d) for the DB approach) and pairwise comparing them to unoptimized versions (such as to Figures 4.4(a), 4.4(c), 4.5(a), and 4.5(e), respectively), we can state the following:

(a) *Local search based techniques can generally benefit more from parameter passing than strategies based on constraint satisfaction.*

(b) *Any increase in the number of bound nodes, which also corresponds to the number of parameters being passed, has a direct speed-up effect for the pattern matching.*

Effects of parallel rule execution. The effect of parallel rule execution (as presented in Fig. 5.9(d)) is noticable in case of all tools that support this feature, but significant speed-up is produced only by PROGRES and the DB approach.

Speed-up effects of parallel rule execution can be explained by the fact that the matching process resumes from the last matching, while the iterative approach repeats the complete matching procedure from the beginning repeatedly examining already processed partial matchings before finding a new one. The DB approach can also benefit from the reduction of average time needed for overhead actions ensuring the consistency of models. For instance, query plans have to be generated and table rows to be modified need to be locked in each graph transformation step. This can be avoided in parallel rule execution, when a single query plan is enough for the whole pattern matching phase and the influenced rows can be locked in the beginning of the parallel rule application and released in the end (transaction handling).

The speed-up of parallel rule execution is mainly caused by the fact that parallel rewriting loops through all matchings of the rule in a single sweep, while the iterated approach restarts the pattern matching process for each rule application and examines first all already processed nodes before finding a new matching. As a result, the average execution time is reduced by a factor depending on the number of matchings.

5.6 Conclusion

In the current chapter, I proposed a benchmarking framework, which enables quantitative performance analysis and comparison of graph transformation tools and their optimization strategies.

- *Definition and categorization of the features of graph transformation problems.* By analyzing typical scenarios described by graph transformation, I defined and categorized such characteristics of graph transformation problems that have high influence on tool performance such as pattern size, the maximum degree of nodes, the number of matchings and the length of transformation sequences (Sec. 5.2.2).
- *Identification and categorization of the optimization strategies in graph transformation tools.* By analyzing the most popular graph transformation tools, I identified and categorized their typical optimization strategies such as parallel rule execution, 'as long as possible' rule application, multiplicity based optimization, and parameter passing, which have significant impact on execution time (Sec. 5.2.3).
- *Specification of benchmark examples.* I adapted standard benchmarking terminology for graph transformation, and I prepared benchmark example specifications for typical model interpretation and model transformation scenarios (Sections 5.3 and 5.4).
- *Quantitative comparison of the speed-up effects of optimization strategies in graph transformation tools.* Based on the benchmarking framework, I carried out measurements on several tools by using different parameter settings and combinations of optimization strategies. I used the measurement results for assessing the acceleration effects of optimization strategies (Sec. 5.5).

The results of this chapter are based on [55, 147, 153, 154].

Relevance

As stated in [123], creation and widespread use of benchmarking within a research area is frequently accompanied by rapid technical progress and community building. Our optimistic vision is to achieve the same in the graph transformation community. The results of the current chapter can be considered as the first step on this way, as our benchmarking framework is the first approach that provides a systematic method for assessing the performance of graph transformation tools.

Although it is early to make any strict statements on the overall role of our benchmarking framework in the process of community building, both the specifications and our published measurement results have already been used by the developers of GrGen in [11, 12, 51, 132] to determine the performance behaviour of their tool. Additionally, this benchmarking framework has been used for the performance analysis of the techniques and algorithms to be presented in upcoming chapters of the current thesis.

The proposed approach can obviously be extended by preparing new benchmark examples (e.g., for the model analysis scenario) and by involving further graph transformation tools in the measurements to provide a wider range comparison to the community. Additionally, in [52], developers of GrGen proposed several improvements for the benchmarking framework on chronometry, evaluation technique and rule design optimization issues, which can be considered as a first step in the community building process.

The presented benchmarking framework has several graph transformation independent issues and characteristics, which have also been identified and handled in benchmark approaches originating from other fields of computer engineering. These common topics include the most basic concepts of benchmarking, namely, to carry out repeatable performance measurements in a precisely defined environment, the methods of chronometry, and several operating system related issues like the time-sharing of process executions, which might worsen the precision of measured values.

The experience gained from measuring the performance of tools can be used in the improvement of pattern matching engines and their optimization strategies as in case [50] reported by the FUJABA developers and in the development of model transformation tools that use declarative and rule-based specification languages (like the Relations and Core languages of QVT). The latter direction should be highly emphasized as for QVT, only initial prototypes (e.g., MTF [2]) have been developed, and efficient product quality implementations are missing.

Further possible future tasks are to study the differences between navigation and constraint checking based pattern matching approaches, to examine the performance effects of combining optimization strategies, and to adapt and repeat our measurements and analysis for simulation and model transformation tools.

Limitations

The presented graph transformation benchmarking framework has several known limitations. First of all, its direct applicability as a general model transformation framework is limited to those approaches whose implementation is based on graph transformation. By considering the benchmarking framework in a broader sense, this applicability restriction can be weakened to such transformations that can be specified by declarative and rule-based transformation languages, which criterion is obviously fulfilled by the tools based on the Relations and Core languages of the QVT standard. However, this extension can be a non-trivial task due to the large variety of essentially different implementation techniques [30] used in model transformation tools.

Since the basic structure of graph transformation rules and QVT rules is similar, one possible way to use the benchmarking framework for model transformations is to measure execution times with rule application granularity without considering the separate handling of pattern matching and updating phases. This strategy can be realized in a rather straightforward way by inserting codes with time measuring functionality before and after the invocation of rule executions. However, note that even this approach might easily fail in situations when transformation rules can call each other as the current version of the benchmarking framework does not handle this case.

From graph transformation aspects, the strong focus on the pattern matching phase, and the rewriting of a single model can be considered as limitations of the framework. Furthermore, the current quantitative analysis is limited to the isolated testing of different optimization strategies. Only the combination of multiplicity based optimization and parameter passing is concerned in the presentation of raw measurement results.

CHAPTER

6 Graph Transformation in Relational Databases

In this chapter, I present a novel technique (referred previously as the DB approach) for implementing graph transformation based on standard relational database management systems (RDBMSs). As a result, a robust and fast transformation engine can be obtained, which is especially suitable for extending modeling tools with an underlying RDBMS repository and embedding model transformations into large distributed applications where models are frequently persisted in a relational database and transaction handling is required to handle large models consistently.

6.1 Motivation

As mentioned in Section 1.4, the transformation of huge, industrial size system models had never been investigated despite the fact that all the state-of-the-art graph transformation tools in 2003 performed in-memory translations, which suffered from memory shortage, when models of such size were processed. A possible solution to overcome the problem of exceeding main memory limits is to perform all the calculations on models stored on *disks*, which enables the handling of larger models for the price of slower runtime operations. This idea can be realized by building graph transformation on top of a relational database.

Additionally, several large distributed applications already exist in the software industry, in which models are persisted in a relational database. If users aim at embedding model transformation support into such systems, RDBMS based graph transformation can provide an implementation for this task.

Relational database management systems that serve as the storage medium for business critical data for large companies are probably the most successful products of software engineering. A crucial factor in this success is the close synergy between theory and practice. The Structured Query Language (SQL) [49], which is built upon the precise mathematical foundations of relational algebra, enables declarative specification for defining, manipulating and querying data in RDBMSs.

Since graph transformation rules can also be considered as a declarative specification for manipulating graph-based models, the integration of graph transformation and relational database techniques can improve the robustness of GT engines, which are built on top of RDBMSs by introducing transaction support, by providing a formal background for verifying the correctness of rule applications, and by always executing precise and well-founded operations on the database.

Related work

Intensive research has already been carried out for integrating graph transformation and relational database techniques. However, these approaches have been focusing on how graph transformation could be adapted as a visual query and data manipulation language for databases. The following list is a brief selection of some main results in the field.

- In [3], a hybrid (visual and textual) query language is proposed together with a method, which translates hybrid queries into traditional textual queries by graph transformation.

  In this approach, the graphical part of hybrid queries is based on an Entity–Relationship (E/R) diagram notation, while the target (textual) language is an object-relational extension of SQL. The PROGRES tool [122] has been employed for graph transformation. Generated SQL queries use the concept of subqueries for expressing restrictions posed by the graph structure.

- [67] proposes the use of triple graph grammars [120] for database re-engineering of legacy systems in their Varlet framework. PROGRES has been used again as a graph transformation engine, and it translates the database schema described by an E/R diagram to an object-oriented conceptual model.

It is common in all these approaches that they investigate how graph transformation can contribute to object-relational database design or to other database related tasks, such as translating hybrid queries to textual ones. Another common feature is that they all use a graph-oriented underlying database (namely GRAS).

In contrast to the above approaches, our proposal is to examine how databases can potentially contribute to the paradigm of graph transformation.

This direction has already been examined at the University of Aachen, where GRAS [75] has been developed. GRAS is a graph-oriented database management system, which served as the underlying database for the PROGRES [122] graph transformation tool.

On one hand, GRAS shares many functionalities with relational databases. Namely, it has a query language, and it supports transaction handling, and consistency checking. On the other hand, GRAS operates on an object-oriented data model (based on attributed graphs), instead of the relational data model used in our approach. Recent versions of the GRAS database (namely GRAS/GXL and DRAGOS [17]) can already access underlying RDBMSs, but these versions still provide a graph based interface to transformation tools (e.g., PROGRES), and in this sense, GRAS can be considered as a highly sophisticated object-relational mapping layer.

From the application area aspects, GRAS provides a graph oriented model repository with many advanced features for graph transformation tools, while our approach focuses on to provide a model transformation plugin for legacy systems that already store models in relational databases.

Our choice for using a plain relational DBMS as an underlying database instead of object-oriented or object-relational DBMSs was motivated by its large popularity in practical applications, the wide range of implementations, and the mature theory based on the relational data model.

Objectives

I propose an approach to implement graph transformation built on top of standard relational database management systems (RDBMSs). The essence of the approach is to create database views for each rule and to handle pattern matching by inner join operations while handling negative application conditions by left outer join operations. Furthermore, the model manipulation prescribed by the application of a

graph transformation rule is also implemented using elementary data manipulation statements (such as insert, delete).

Furthermore, I implemented a prototype graph transformation engine, which uses open, off-the-shelf relational databases (such as PostgreSQL [97] or MySQL [125]) as a backend to demonstrate the practical feasibility of the proposed approach. For a detailed experimental evaluation, I assess how the performance of the prototype is influenced by parallel rule applications, RDBMS-specific query optimization techniques, and the choice of the underlying RDBMS.

Finally, I propose a database independent and portable GT engine implementation that uses the declarative queries of the EJB QL standard in the pattern matching phase instead of SQL commands, which aims at resolving the diversity of SQL dialects using standard J2EE technology.

Structure

The basic structure of the current chapter is the following.

- Section 6.2 informally summarizes the essence of our approach on an example prior to going into deep mathematical details. This section assumes a basic knowledge of relational database concepts.
- Section 6.3 surveys the main concepts of relational databases together with their formal definitions.
- Section 6.4 presents the formalization of our approach to encode graph transformation rules into relational databases. Formal proofs of correctness are listed in Appendix A.
- Section 6.5 investigates how the performance of graph transformation over a RDBMS depends on the selection of the underlying database, on the application of the built-in query optimizer, and on the usage of the parallel rule execution tool feature.
- Section 6.6 presents a portable, database independent GT engine implementation that uses queries expressed on the Enterprise Java Beans Query Language (EJB QL) for specifying graph pattern matching.
- Section 6.7 concludes the current chapter with emphasizing its relevance.

6.2 Graph transformation in relational databases: An informal overview

An informal overview is provided on how graph transformation rules can be implemented by using traditional relational database techniques. Concepts are presented on the running example of object-relational mapping, which has already been introduced by Example 1 in Sec. 2.2, and by Example 5 in Sec. 3.1. In the approach being presented in this chapter, attribute handling is not discussed in order to preserve the notational consistency of the whole dissertation. However, the implementation is able to handle attributes as shown by [151].

Mapping metamodels to database tables. In the first step, a standard mapping (for more details see [49, 110]) is used to generate the schema of the database from the metamodel.

- Each class with k outgoing many-to-one associations is mapped to a table with $k + 1$ columns. Column id will store the identifiers of objects of the specific class. All other columns will contain the identifiers of target objects of such outgoing many-to-one links that have the corresponding association as their direct type. If no such outgoing link exists in the model, the undefined (NULL) value is used in the corresponding column. Additional foreign key constraints, whose

role is to guarantee the consistency of the database have to be defined for columns representing many-to-one associations referring to the table assigned to the corresponding target class.

- A table with 2 columns storing the identifiers of source and target objects is assigned to each many-to-many association. Additionally, foreign key constraints are defined for both columns referring to the tables assigned to the corresponding source and target class, respectively.

- Inheritance is handled by a foreign key constraint defined for the identifier column id of the table assigned to the subclass. This foreign key constraint maintains reference to the identifier column id of the superclass table.

Database representation of instance models. Instance models representing the system under design are stored in these database tables.

- A unique identifier is assigned to each object of the instance model.

- The identifier of each object has to appear in the column id of all tables that correspond to ancestors of the object's direct type.

- The database representation of a many-to-one link is a row in the table that corresponds to the source class of the link's type. This row should contain the identifiers of source and target objects in the identifier column id and the column representing the many-to-one association, respectively.

- Each many-to-many link is represented in the database by a pair of source and target object identifiers appearing in the table that corresponds to the direct type of the link.

Example 16 The model of Fig 5.8(d) (also shown in Fig. 6.1(b)) has been selected as an instance model for the current running example, for which the corresponding database representation is depicted in Fig. 6.1(c).

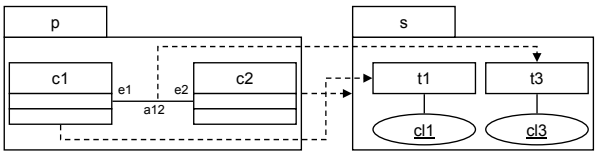
- **A sample database representation of an object.** The model of Fig. 6.1(b) contains a UML class c1, which is identified by the key c1 in the database. As ModelElement, Namespace and Class are ancestors of Class according to the metamodel of Fig. 2.1, all their corresponding tables should have the key c1 in their identifier column id.

- **A sample database representation of a many-to-one link.** UML class c1 is contained by UML package p. This containment is a many-to-one link of type EO going from UML class c1 to UML package p. The database representation of this link is a row in the ModelElement table, which has values c1 and p in columns id and EO, respectively.

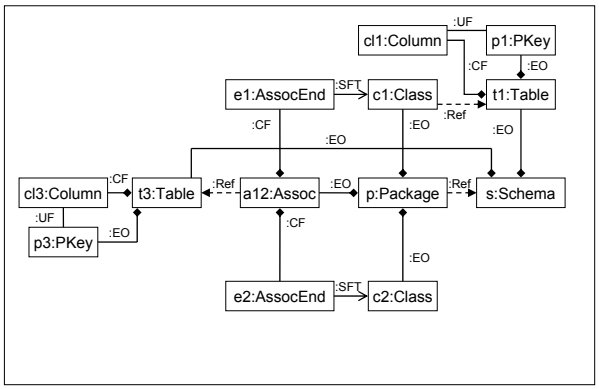
- **A sample database representation of a many-to-many link.** The many-to-many link of type UF connecting primary key p3 and column cl3 is represented by a corresponding row in table UF.

Views for LHS and NAC. The matching patterns of a graph transformation rule are calculated by using views, which contain all matchings of the rule. More specifically, we introduce a separate view for each LHS and NAC graph.

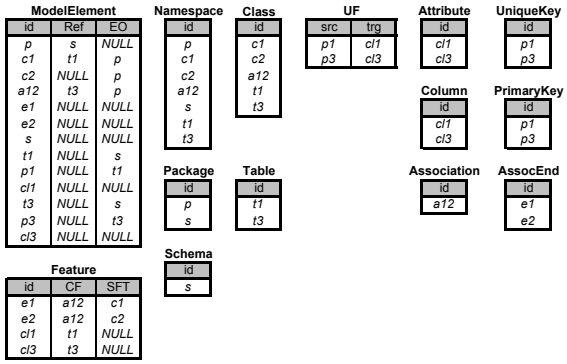
(a) The view generated for rule graphs (LHS and NAC) executes an *inner join operation* on tables that represent either a node or an edge of the rule graph.



(a) Concrete syntax of the instance model of Fig. 6.1(b)



(b) The instance model of Fig. 5.8(d)



(c) The corresponding database representation

Figure 6.1: A sample instance model and its corresponding database representation

(b) The joined table is *filtered by injectivity and edge constraints*. Injectivity constraints express the injective mapping of rule graph nodes and edges on the database level. Edge constraints define restrictions imposed by the graph structure, which means that the source (target) node identifier of the given edge should be found in tables representing the type of the edge and the type of the source (target) node.

(c) Finally, a *projection* selects only those columns of the filtered joined table that represent node identifiers. Information about the source and target nodes of edges is discarded during projection. This information is unnecessary in the sequel, since requirements imposed by the graph structure have already been checked and fulfilled.

Example 17 The essence of this approach is introduced by an example listing the view generated for the LHS and NAC pattern of ClassRule (see Fig. 3.1(b)).

```
CREATE VIEW ClassRule_lhs AS                -- an LHS view
SELECT c.id AS c, p.id AS p, s.id AS s      -- with 3 columns
FROM Class AS c, ModelElement AS c_anc,
     Package AS p, ModelElement AS p_anc,
     Schema AS s
WHERE c.id = c_anc.id AND c_anc.EO = p.id   -- EO edge eo1
  AND p.id = p_anc.id AND p_anc.Ref = s.id  -- Ref edge r1
  AND p.id <> s.id                          -- injectivity constraint
                                            -- for nodes p and s

CREATE VIEW ClassRule_nac AS
SELECT c.id AS c, tn.id AS tn
FROM Class AS c, ModelElement AS c_anc,
     Table AS tn
```

```
WHERE c.id = c_anc.id AND c_anc.Ref = tn.id -- Ref edge rn
  AND c.id <> tn.id                        -- injectivity constraint
                                           -- for nodes c and tn
```

The LHS of ClassRule requires the presence of an EO edge that connects a UML class to a UML package. Since EO edges are stored in the ModelElement table, it must also be included in the inner join operation in addition to tables Class and Package. Since the source node of eo1 has to be a UML class, only such source object identifiers of the column id of table ModelElement can participate in a matching that can also be found in table Class as expressed by the edge constraint `c.id = c_anc.id`. A similar edge constraint `c_anc.EO = p.id` requires possible target object identifiers of column EO in table ModelElement to be equal to a value from the identifier column of table Package. A similar pair of equalities express the edge constraints for the reference edge r1. Due to inheritance relations defined in the metamodel, every schema is a UML package at the same time. Thus, pattern nodes S and P are not allowed to be mapped to the same object. On the database level, this (injectivity) constraint is expressed by the inequality `p.id <> s.id`.

ClassRule_lhs

c	p	s
c1	*p*	*s*
c2	*p*	*s*
a12	*p*	*s*

ClassRule_nac

c	tn
c1	*t1*
a12	*t3*

ClassRule_left_join

c	p	s	tn
c1	*p*	*s*	*t1*
c2	*p*	*s*	*NULL*
a12	*p*	*s*	*t3*

ClassRule

c	p	s
c2	*p*	*s*

Figure 6.2: Database representation of matchings

The left part of Fig. 6.2 shows the contents of views that have been defined for the LHS and the NAC parts of ClassRule.

For instance, c1 is a UML class in the UML package p and this UML package is connected to schema s by a reference edge in Fig. 6.1(b), thus, a matching for the LHS of ClassRule is found, which is represented by a corresponding row in the leftmost view of Fig. 6.2. Note that the LHS of ClassRule has 2 further matchings as shown by the 2 additional rows in the same view.

Since UML class c1 is connected to table t1 by a reference edge in the model of Fig. 6.1(b), the view generated for the NAC contains a corresponding row for this matching. This view has one further row for representing the other matching.

Left joins for preconditions of rules. When the view for the precondition graph is calculated, views of all its positive and negative application conditions are available. If the precondition has no negative application conditions then the view defined for the LHS contains the database representation of all matchings of the precondition graph.

(a) Each NAC view is *left outer joined* to the LHS view one by one. The *join condition* of this operation expresses that columns representing the same shared node in the LHS and the NAC graphs should be equal.

(b) For a matching of the precondition graph, we require (in the *null condition*) that columns of NAC(s), which are shared with the LHS part, are filled with undefined values. This means that there are no possible extensions of a matching of the LHS that is also a matching of (any) NAC graph.

(c) Then a *projection* is performed, which displays only those columns that originate from LHS.

Example 18 To continue our running example, we present the view definition for the precondition of ClassRule.

```
CREATE VIEW ClassRule AS
SELECT lhs.*
FROM ClassRule_lhs AS lhs
   LEFT JOIN ClassRule_nac AS nac ON lhs.c = nac.c
WHERE nac.c IS NULL
```

The left part of Fig. 6.2 shows the contents of views that have been defined for the LHS and the NAC of ClassRule, respectively. The third table of Fig. 6.2 presents the result of the left outer join operation, while the last table corresponds to the precondition of ClassRule. Note that columns representing UML class C are shared between LHS, NAC graphs, so these columns appear both in the join and in the filtering condition.

After executing the left outer join operation, the result has 3 rows. Since rows with values c1, and a12 in column c can be found in both LHS and NAC views, the corresponding 2 rows in the left outer joined table are completely filled. On the other hand, the second row of the LHS view (i.e., with value c2 in column c) has no corresponding row in the NAC view. As a consequence, the left outer joined table has NULL value in column tn of its second row. As this is the only row that is not filtered out by the null condition, it can also be found in the view generated for the whole precondition graph, which means that a single matching has been found for ClassRule, and as a consequence, the rule is applicable on that matching.

Model manipulation in relational databases. Operations in the graph manipulation phase can be implemented by issuing several data manipulation commands (INSERT, DELETE, and UPDATE) in a single transaction block. The transaction block is needed to ensure that a graph transformation step is atomic, i.e., either all commands or none of them are executed to result in a consistent model after rule application.

In the graph manipulation phase, deletions are followed by insertions.

- We further restrict the order of delete operations in such a way that edge deletions precede node deletions.
  - If a many-to-one link has to be deleted from the model, then the table that represents the source class of the direct type association of the given link has to be updated. Specifically, the value of the column corresponding to the many-to-one association has to be set to NULL in the row that contains the source node identifier of the link in its column id.
  - In case of a deletion of a many-to-many link, the row consisting of the source and the target node identifiers of the link has to be removed from the table that corresponds to the direct type of the given link.
- As the node identifier to be deleted can be found in tables representing the ancestors of the object's direct type, the deletion should proceed in a bottom-up order (to respect foreign key constraints) by starting at the class, which is the direct type of the object.

  During this iteration, additional attention is needed to consistently handle the removal of dangling edges from the database. As a first step, all associations have to be determined, whose source or target is the class, which is just being traversed by the iteration. Then we should perform the above mentioned edge deletion procedure on all links that have the object to be deleted as their

source or target node and that are instances of associations collected in the previous step. The final step of the iteration is the deletion of the object itself from the table that corresponds to the class being traversed. This is performed by deleting the row of this table, which contains the identifier of the given object in its column id.

For handling node and edge insertions on the database level in the graph manipulation phase, we can use exactly the same procedures as for the initial table filling phase.

We state that the new content of database tables always corresponds to the derived model, thus it can be proven that our approach performs graph transformation over an underlying relational database.

Example 19 We continue our sample graph transformation rule ClassRule with the model manipulation parts. This rule prescribes the insertion of a new table that contains a single column with a primary key. In addition, one many-to-many link and four many-to-one links have to be added to the model as specified by Fig. 3.1(b).

On the database level, the same effect can be achieved by generating new identifiers t2, p2, and cl2 for the new table, primary key, and column, respectively. For instance, identifier t2 is inserted into all tables that represent the ancestors of Table. Identifiers of other new objects such as p2 and cl2 are handled similarly. In order to respect foreign key constraints, insertions are executed in a top-down order starting at the table corresponding to the most general ancestor. Insertion of the 4 new many-to-one links appears as the 4 update operations presented in the listing below. Finally, the new many-to-many link of type UF is added to the database by executing the corresponding insert operation.

```
-- Creating table t2
INSERT INTO ModelElement (id) VALUES (t2);
INSERT INTO Namespace (id) VALUES (t2);
INSERT INTO Class (id) VALUES (t2);
INSERT INTO Table (id) VALUES (t2);
-- Creating primary key p2
INSERT INTO ModelElement (id) VALUES (p2);
INSERT INTO UniqueKey (id) VALUES (p2);
INSERT INTO PrimaryKey (id) VALUES (p2);
-- Creating column cl2
INSERT INTO ModelElement (id) VALUES (cl2);
INSERT INTO Feature (id) VALUES (cl2);
INSERT INTO Attribute (id) VALUES (cl2);
INSERT INTO Column (id) VALUES (cl2);
-- Creating 5 links
UPDATE ModelElement SET Ref = t2 WHERE id = c2;
UPDATE ModelElement SET EO = s WHERE id = t2;
UPDATE ModelElement SET EO = t2 WHERE id = p2;
UPDATE Feature SET CF = t2 WHERE id = cl2;
INSERT INTO UF (src,trg) VALUES (p2,cl2);
```

When the execution of these graph manipulation commands terminates, the new content of database tables corresponds to the derived model of Fig. 5.8(e).

6.3 Database operations

In our graph transformation engine a relational DBMS is used to represent metamodels as database schemas, to store instance models and to perform modifications on such models. Now we summarize the database terminology used throughout this chapter.

6.3.1 Tables and views

The most basic entities of a database are tables that may have several columns and their role is to store data in its rows.

Definition 29 A **database table with n columns** (denoted by $\mathcal{T}^{(n)}(A_1, \ldots, A_n)$) is an n-ary relation over sets $(C_1 \cup \{\varepsilon\}), \ldots, (C_n \cup \{\varepsilon\})$. $\mathcal{T}$ and A_i denote names of the table and of the ith column, respectively. Column names definitely have to be unique in the scope of a single table, thus a table cannot have columns sharing the same name. The ith column of the table may contain values from the set C_i. Undefined (or null) values (denoted by ε) are also allowed in any columns. Formally, $\mathcal{T}(A_1, \ldots, A_n) \subseteq (C_1 \cup \{\varepsilon\}) \times \ldots \times (C_n \cup \{\varepsilon\})$.

Definition 30 Since database tables are n-ary relations, their elements are n-tuples $\vec{x} = (x_1, \ldots, x_n)$, which are called **rows** in database terminology.

While the traditional relational DBMSs use multi-set semantics, we can simplify to set semantics in the paper, since uniqueness of rows can be guaranteed by the algorithm that will be presented in Sec. 6.4.

Definition 31 A **direct column reference for a table** $\mathcal{T}$ (denoted by $\mathcal{T}.A_i$ or simply by A_i (if the table to which it refers can unambiguously be determined)) identifies the column of $\mathcal{T}$ that has a name A_i.

Definition 32 Given a table $\mathcal{T}$ with a column called A_i, a **direct column reference for a row** $\vec{t} \in \mathcal{T}$ (denoted by $\vec{t}[A_i]$) identifies the element of $\vec{t}$ that can be found in the column $\mathcal{T}.A_i$.

Definition 33 A **primary key constraint for columns** $A_1, \ldots, A_j$ **of table** $\mathcal{T}(A_1, \ldots, A_n)$ guarantees the uniqueness of values in the selected set of columns. Formally, $\forall \vec{r}, \vec{s} \in \mathcal{T} : (\vec{r} = \vec{s} \iff \forall i,\ 1 \leq i \leq j\ :\ \vec{r}[A_i] = \vec{s}[A_i])$, where $\vec{r}[A_i]$ and $\vec{s}[A_i]$ (see Def. 32) refer to the elements in column A_i of rows $\vec{r}$ and $\vec{s}$, respectively.

Foreign key constraints are integrity constraints provided by the most RDBMSs. Their role is to ensure that columns in different tables never contain inconsistent data. In our approach, these constraints are (mainly) used to guarantee that the database representation of an edge can never appear in the database without its source and target nodes being already present.

Definition 34 A **foreign key constraint for column** $\mathcal{R}.A$ **referring to column** $\mathcal{S}.B$ (denoted by $\mathcal{R}.A \overset{FK}{\rightarrow} \mathcal{S}.B$) declares that all values of column $\mathcal{R}.A$ should also be found in column $\mathcal{S}.B$, or formally $\mathcal{R}.A \subseteq \mathcal{S}.B$.

Definition 35 A **view** $\mathcal{V}$ is a relation calculated by applying the query operations of Sec. 6.3.2 on tables.

Definition 36 The **database schema** (denoted by $\mathfrak{S}_{DB}$) consists of the set of tables and views appearing in the database.

6.3.2 Query operations

After introducing the basic entities (i.e., tables), query operations are discussed, which can be used to define derived tables (i.e., views).

Definition 37 Given an ordered sequence of column references $\mathcal{T}.A_1, \ldots, \mathcal{T}.A_k$ for $\mathcal{T}$, the **projection of a table $\mathcal{T}$ to columns** $A_1, \ldots, A_k$ (denoted by $\pi_{A_1,\ldots,A_k}(\mathcal{T})$) is a k-ary relation, which consists of only the enumerated columns of $\mathcal{T}$. Its formal definition is as follows

$$(x_1, \ldots, x_k) \in \pi_{A_1,\ldots,A_k}(\mathcal{T}) \iff \exists (y_1, \ldots, y_n) \in \mathcal{T} : \bigwedge_{i=1}^{k} x_i = y_{A_i},$$

where $\bigwedge_{i=1}^{k} x_i = y_{A_i}$ denotes the conjunction (logical AND) of equalities.

In SQL terms projection is implemented in the select statement as follows:

```
SELECT A1,...,Ak FROM R;
```

Definition 38 An **atomic expression** has a form $\alpha\theta\beta$, where α and β can be either a column of $\mathcal{T}$ or a constant c. θ is a comparison operator, so $\theta \in \{=, <, >, \leq, \geq, \neq\}$. A **formula** F is either an atom or it is constructed from atoms using the logical and ($\wedge$), logical or ($\vee$), and negation ($\neg$) operators.

Definition 39 Given a formula F, **selection** (denoted by $\sigma_F(\mathcal{T})$) operates on a single table $\mathcal{T}$ and collects the rows of $\mathcal{T}$ where $F(y_1, \ldots, y_n)$ holds. The formal definition of selection is

$$\sigma_F(\mathcal{T}) = \{\, (y_1, \ldots, y_n) \mid (y_1, \ldots, y_n) \in \mathcal{T} \wedge F(y_1, \ldots, y_n) = \text{ true} \,\}.$$

An obvious corollary is that $\sigma_F(\mathcal{T}) \subseteq \mathcal{T}$.

Selection operation can also be expressed in SQL, using a WHERE condition with F as its parameter.

Definition 40 The **cross join of tables** $\mathcal{R}^{(m)}$ **and** $\mathcal{S}^{(n)}$ (denoted by $\mathcal{R} \times \mathcal{S}$) is a table with $m + n$ columns and it is the Cartesian product of the two tables. A row is in the result table, if its first m values correspond to a row in $\mathcal{R}$ and its last n values corresponds to a row in $\mathcal{S}$. Its formal definition is:

$$\mathcal{R} \times \mathcal{S} = \{\, (x_1, \ldots, x_m, y_1, \ldots, y_n) \mid (x_1, \ldots, x_m) \in \mathcal{R} \wedge (y_1, \ldots, y_n) \in \mathcal{S} \,\}.$$

Cross join operation also exists in SQL, which can be formulated as:

```
SELECT * FROM R,S;
```

Column name uniqueness has only a table scope, so name clashes may occur in joint tables. In order to avoid this uncomfortable consequence caused by join operations, we should be able to differentiate between columns that originate from different base tables.

In RDBMSs name clashes are resolved by some renaming mechanisms. The SQL notation for renaming depends on the actual RDBMS software that is being used. In this paper, we use the PostgreSQL notation, namely the `AS` keyword for this purpose in SQL queries (e.g., `T.id AS T`). In our mathematical formalism, column sets implement the table renaming functionality, while column renaming is performed implicitly by defining a new name for a column in the view definition.

Definition 41 Given two tables $\mathcal{R}^{(m)}$ and $\mathcal{S}^{(n)}$, a **column set of a joint table** $\mathcal{R} \times \mathcal{S}$ **referring to the base table** $\mathcal{R}$ (denoted by R^{cs}) is the largest possible set of columns that originate from table $\mathcal{R}$, which is the first m columns of $\mathcal{R} \times \mathcal{S}$ in this case.

Definition 42 Given two tables $\mathcal{R}$ and $\mathcal{S}$, an **indirect column reference for the joint table** $\mathcal{T} = \mathcal{R} \times \mathcal{S}$ (denoted by $\mathcal{T}.\mathcal{R}^{cs}.A_i$, or simply by $\mathcal{R}^{cs}.A_i$) identifies a column of $\mathcal{T}$ by selecting a column set first and then by using the direct column reference A_i on the column set.

An indirect column reference for a row of the joint table can be similarly defined.

Definition 43 Given a formula F, the **inner join of tables** $\mathcal{R}$ **and** $\mathcal{S}$ (denoted by $\mathcal{R} \overset{F}{\bowtie} \mathcal{S}$) is a selection from the Cartesian product filtered by formula F. Formally,

$$\mathcal{R} \overset{F}{\bowtie} \mathcal{S} = \sigma_F(\mathcal{R} \times \mathcal{S}).$$

In this paper, only atoms of type $A = B$ (two column names in equality relation) and the logical AND operator will be used for basic atoms and for constructing formulae, respectively. Typically, A and B are taken from different tables. It is useful from a practical point of view, if column names on the different sides of the equality relation are from different tables. However, the general definition does not require any such restrictions. SQL notation of the inner join operation is as follows.

```
SELECT * FROM R INNER JOIN S ON R.A=S.B;
```

Definition 44 Given a formula F, the **left outer join of tables** $\mathcal{R}$ **and** $\mathcal{S}$ (denoted by $\mathcal{R} \overset{F}{\ltimes} \mathcal{S}$) (i) contains all the rows of $\mathcal{R} \overset{F}{\bowtie} \mathcal{S}$, (ii) additionally contains all such rows of $\mathcal{R}$, for which there does not exist any row in $\mathcal{S}$, where $F(\vec{x}|\vec{y})$ holds, and (iii) the latter rows are filled with undefined values in columns originating from $\mathcal{S}$.

The formal definition of left outer join is

$$\mathcal{R} \overset{F}{\ltimes} \mathcal{S} = (\mathcal{R} \overset{F}{\bowtie} \mathcal{S}) \cup \{\, (\vec{x}, \varepsilon, \ldots, \varepsilon) \mid \vec{x} \in \mathcal{R} \wedge \nexists \vec{y} \in \mathcal{S} \text{ for which } F(\vec{x}|\vec{y}) = true \,\}.$$

where $F(\vec{x}|\vec{y})$ denotes whether formula F is satisfiable if its unbound variables are replaced by the corresponding values of rows $\vec{x}$ and $\vec{y}$.

A sample query presenting the left outer join operation is

```
SELECT * FROM R LEFT JOIN S ON R.A=S.B;
```

6.3.3 Data manipulation operations

Finally, we define three data manipulation operations. $\mathcal{T}'$ will mark the content of table $\mathcal{T}$, after the database operation has completed.

Definition 45 The **delete operation**

$$\texttt{DELETE FROM } \mathcal{T} \texttt{ WHERE } A_1 = y_1 \texttt{ AND } \ldots \texttt{ AND } A_k = y_k$$

removes those rows of table $\mathcal{T}$, which contain values y_i in their column A_i, respectively. Formally, $\mathcal{T}' = \mathcal{T} \setminus \left\{ \vec{x} \in \mathcal{T} \mid \bigwedge_{i=1}^{k} \vec{x}[A_i] = y_i \right\}$, where $\bigwedge_{i=1}^{k} \vec{x}[A_i] = y_i$ denotes the conjunction (logical AND) of equalities.

Definition 46 The **update operation**

$$\texttt{UPDATE } \mathcal{T} \texttt{ SET } A_j = y \texttt{ WHERE } A_i = x$$

sets the value of column A_j to y in all rows of table $\mathcal{T}(A_1, \ldots, A_n)$ where column A_i has value x. Formally, $\mathcal{T}' = (\mathcal{T} \setminus Minus) \cup Plus$, where

$$Minus = \{\, \vec{z} \in \mathcal{T} \mid \vec{z}[A_i] = x \,\}$$

and

$$Plus = \left\{ \vec{z}_{new} \mid \exists \vec{z} \in Minus, \forall k \in \mathbb{Z}_n^+ \, : \, \vec{z}_{new}[A_j] = y \wedge \bigwedge_{j \neq k} \vec{z}_{new}[A_k] = \vec{z}[A_k] \right\},$$

where $\mathbb{Z}_n^+$ denotes the set of positive integers up to n (i.e., $1 \leq k \leq n$).

Definition 47 The **insert operation**

$$\texttt{INSERT INTO}\ \mathcal{T}\ (A_1, \ldots, A_k)\ \texttt{VALUES}\ (y_1, \ldots, y_k)$$

adds an n-tuple $\vec{y}$ to table $\mathcal{T}$, if $\vec{y}$ is not yet contained. The tuple $\vec{y}$ has value y_i in column A_i, respectively, and it contains undefined values in all other columns. In other words, $\mathcal{T}' = \mathcal{T} \cup \{ \vec{y} \}$, where $\vec{y}[A_i] = y_i$, if $1 \leq i \leq k$, and $\vec{y}[C] = \varepsilon$, if $C \notin \{ A_1, \ldots, A_k \}$.

Definition 48 Given a sequence of database operations TA, a **transaction is executed on a representation $\mathfrak{M}$ resulting in an other representation** $\mathfrak{M}'$ (denoted by $\mathfrak{M} \overset{TA}{\Longrightarrow} \mathfrak{M}'$), if either all operations of TA or none of them are executed.

6.4 Graph transformation in relational databases

We formally present how a graph transformation engine (following the single pushout [118] approach with injective matchings) can be implemented using a relational database. First, we describe how an appropriate database schema can be created based on the metamodel, and how the database representation of the model can be generated (Sec. 6.4.1). Afterwards, the pattern matching phase of rule application is implemented using database queries (Sections 6.4.2 and 6.4.3), finally data manipulation is handled (in Sec. 6.4.4).

6.4.1 Mapping metamodels and models to database tables

Mapping of metamodels to database tables. Instance models representing the system under design are stored in database tables. We use the standard bi-directional mapping (for more details see [49, 110]) to generate the schema of the database with BCNF property [27] from the metamodel.

- Let us first introduce a set called *database identifier universe* (denoted by $\mathfrak{U}^d$), which denotes the set of all identifiers that might be stored in the database.
- Each class C with k outgoing many-to-one associations ($C \overset{A_1}{\rightarrow}_1 C_1, \ldots, C \overset{A_k}{\rightarrow}_1 C_k$) is mapped to a table with $k+1$ columns $C^d(id, A_1^d, \ldots, A_k^d)$.
  - Column id will store the identifiers of objects of the specific class.
  - Column A_i^d will contain the identifiers of target objects of such outgoing many-to-one links that have association $C \overset{A_i}{\rightarrow}_1 C_i$ as their direct type. If no such outgoing link exists in the model, the undefined value ε is used in column A_i^d.

  Additionally, we should define foreign keys $\forall i \in [1..k] \, : \, C^d.A_i^d \overset{FK}{\rightarrow} C_i^d.id$ to respect the graph structure in the database. Formally, $C^d \subseteq \mathfrak{U}^d \times \left(C_1^d \cup \varepsilon\right) \times \ldots \times \left(C_k^d \cup \varepsilon\right)$.

- We assign a table $A^d(src, trg)$ for each many-to-many association $C_s \overset{A}{\rightarrow}_* C_t$ connecting classes C_s and C_t in the metamodel. Columns src and trg contain identifiers of source and target objects, respectively. Foreign keys $A^d.src \overset{FK}{\rightarrow} C_s^d.id$ and $A^d.trg \overset{FK}{\rightarrow} C_t^d.id$ should additionally be defined to respect the graph structure (preserve the source and the target of edges) in the database. In a more formal way, $A^d \subseteq C_s^d \times C_t^d$.
- If a class C is inherited from a superclass D, then table C^d should be extended by a foreign key constraint $C^d.id \overset{FK}{\rightarrow} D^d.id$.

We introduced the superscript d to uniformly denote database representations of all kinds of graph transformation related entities. For instance, C^d, r^d_{LHS}, and c^d mark the entities that represent a class C, a rule graph r_{LHS}, and an object c in the database, respectively. This notation is always used as a bi-directional mapping meaning that, e.g., C^d unambiguously identifies the database table that was assigned to class C, and vice versa.

Mapping of instance models into rows. Now we define a bijective mapping, which assigns an identifier to each object of the instance model. The image of the mapping c^d will be used as a primary key that identifies object c in the database.

In order to appropriately represent an object in the database, its key has to be contained by all tables that are assigned to an ancestor of the object's type. Since inheritance relation in the metamodel (i.e., the type hierarchy) poses restriction (in the form of foreign key constraints) on exactly the same set of tables, additional care has to be taken when inserting (or deleting) even a single key (identifier). The order that handles insertion correctly is being defined now.

Definition 49 Given a metamodel MM with inheritance relations that are acyclic, a **topological order of a type** t (denoted by $TopologicalOrder(t)$) is such a sequence of the ancestors of t in which a class D cannot appear before an ancestor C in the order, if $C \overset{*}{\leftarrow} D$.

A natural consequence of the definition is that type t is the last element in its topological order.

Definition 50 Given a metamodel MM with inheritance relations that are acyclic, an **inverse topological order of a type** t (denoted by $InverseTopologicalOrder(t)$) is a topological order of t traversed in the opposite order.

A natural consequence of the definition is that type t is the first element in its inverse topological order.

After fixing a certain topological and inverse topological order of a type to be used in the sequel, Algorithm 6.1 derives the database representation of the initial model as follows.

- We suppose that all the tables are initially empty.
- A new identifier c^d is generated for each object c of the instance model M. Then ancestors of the type $t(c)$ of object c are determined and furthermore they are ordered topologically according to the inheritance relation. The ordering is done in a top-down manner, meaning that the "most general" class is enumerated first. (The role of topological ordering is to avoid the violation of foreign key constraints that have already been imposed on database tables.) The final step is to insert the new identifier to all the tables that have been assigned to the enumerated ancestor classes.

Algorithm 6.1 From instance model to its database representation

1: **for all** $c \in V_M$ {For all objects in model M} **do**
2: $c^d := GenerateNewIdentifier()$
3: **for all** $C \in TopologicalOrder(t(c))$ **do**
4: `INSERT INTO` C^d (id) `VALUES` (c^d) {Inserts the new identifier to all ancestor tables}
5: **end for**
6: **end for**
7: **for all** $a \xrightarrow{e}_1 b \in E_M$ {For all many-to-one links in model M} **do**
8: `UPDATE` $src(t(e))^d$ `SET` $t(e)^d = b^d$ `WHERE` $id = a^d$ {Updates the value in column $t(e)^d$ to b^d in the row with identifier a^d}
9: **end for**
10: **for all** $a \xrightarrow{e}_* b \in E_M$ {For all many-to-many links in model M} **do**
11: `INSERT INTO` $t(e)^d$ (src, trg) `VALUES` (a^d, b^d) {Inserts identifiers of end points a and b into the table that corresponds to many-to-many association $t(e)$}
12: **end for**

- For each many-to-one link $a \xrightarrow{e}_1 b$ of the instance model, the row in the table $src(t(e))^d$, which represents the source object a, is updated by replacing the value in column $t(e)^d$ by the identifier b^d of the target object b.
- For each many-to-many link $a \xrightarrow{e}_* b$ of the instance model, the identifiers of the source and target nodes (a^d and b^d) are inserted to the table $t(e)^d$ that has been assigned to the edge type (association) $t(e)$ of link e.

We introduce a new term that formalizes the consistent database representation of an instance model.

Definition 51 Let a metamodel MM, and a database schema $\mathfrak{S}_{DB}$ be given together with the bidirectional mapping d from MM to the tables of $\mathfrak{S}_{DB}$.

A model M and a database representation $\mathfrak{M}$ are consistent ($M \cong \mathfrak{M}$), if

- each object of the instance model is represented in the database by one row in all the tables that have been assigned to ancestors of the node type. Moreover, these rows must contain the identifier of the object in their identifier column id. Formally, $\forall C \in V_{MM}, \forall c \in V_M$: $\Big(C \xleftarrow{*} t(c) \iff \exists \vec{c} \in C^d : \vec{c}[id] = c^d\Big)$,
- each many-to-one link of the instance model is represented in the database by exactly one row in the table that corresponds to the source class of the type of the edge. This single row must contain identifiers of source objects in the identifier column id and target objects in the column corresponding to the direct type of the edge. Formally, $a \xrightarrow{e}_1 b \in E_M \iff \big(\exists \vec{a} \in src(t(e))^d : \vec{a}[id] = a^d \wedge \vec{a}[t(e)^d] = b^d\big)$, and
- the identifiers of source and target nodes of each many-to-many link (edge) of the instance model can be found exactly in the table that corresponds to the type of the edge. Formally, $a \xrightarrow{e}_* b \in E_M \iff \big(a^d, b^d\big) \in t(e)^d$.

Finally, we formulate a theorem, which states that the database representation that has been created by the above-mentioned initialization algorithm is consistent with the initial instance model.

Theorem 1 *The initial instance model M and its database representation $\mathfrak{M}$ are consistent (see Def. 51). Formally, $M \cong \mathfrak{M}$.*

PROOF Proofs of all theorems can be found in Appendix A.

6.4.2 Views for rule graphs (LHS and NAC).

As it is described in Sec. 6.2, the view generated for rule graphs (LHS and NAC) executes an inner join operation on tables that have been assigned to types of nodes and edges appearing in the rule graph. Then the joined table is filtered by injectivity and edge constraints. Finally, a projection selects only those columns of the filtered joined table that represent node identifiers.

Formalization. In order to define pattern matching calculation for an LHS precisely, let us suppose that $n_V = |V_{\mathsf{LHS}}|$ and $n_E = |E_{\mathsf{LHS}}|$. Let us define a total order on the node and edge sets in which nodes precede edges, and let x_i and z_{n_V+j} be the ith node and the jth edge according to this order, respectively.

Now the view ${r^d_{\mathsf{LHS}}}^{(n_V)}$ for the LHS can be calculated as follows:

$$r^d_{\mathsf{LHS}}(ResCols) = \pi_{ProjColRefs}\left(\sigma_{Inj \wedge Edge}\left(\mathcal{T}\right)\right)$$

- First the *Cartesian product of tables* $\mathcal{T}_i$ is calculated. $\mathcal{T}_i$ denotes the table that was assigned to the type of the ith graph object of r_{LHS}. Formally, $\mathcal{T} = \mathcal{T}_1 \times \cdots \times \mathcal{T}_{n_V+n_E}$, where

$$\mathcal{T}_i = \begin{cases} t(x_i)^d, \text{ when } i \leq n_V \text{ and } x_i \in V_{\mathsf{LHS}} \\ src(t(z_i))^d, \text{ when } n_V < i \leq n_V + n_E \text{ and } u_i \xrightarrow{z_i}_1 v_i \in E_{\mathsf{LHS}} \\ t(z_i)^d, \text{ when } n_V < i \leq n_V + n_E \text{ and } u_i \xrightarrow{z_i}_* v_i \in E_{\mathsf{LHS}} \end{cases}$$

- *Edge constraints.* A pair of equations is defined for each edge of LHS. One such pair expresses that the edge is incident to its source and its target node, respectively. (As the database representation of many-to-many and many-to-one links differ from each other, the corresponding pairs of edge constraints have to be obviously different in their structure.) The conjunction of these equations constitute edge constraints $Edge$. Formally,

$$Edge_{one} = \bigwedge \left\{ z^{cs}.id = u^{cs}.id \wedge z^{cs}.t(z)^d = v^{cs}.id \mid u \xrightarrow{z}_1 v \in E_{\mathsf{LHS}} \right\}$$

$$Edge_{many} = \bigwedge \left\{ z^{cs}.src = u^{cs}.id \wedge z^{cs}.trg = v^{cs}.id \mid u \xrightarrow{z}_* v \in E_{\mathsf{LHS}} \right\}$$

  The edge constraint of the view can be expressed as $Edge = Edge_{one} \wedge Edge_{many}$.

- *Injectivity constraints.* Inj are defined for all pairs of LHS nodes, for which the type of one node is an ancestor of the type of the other. The role of injectivity constraints is to ensure the injective mapping of graph objects.

$$Inj = \bigwedge \left\{ x^{cs}_j.id \neq x^{cs}_k.id \mid x_j, x_k \in V_{\mathsf{LHS}} \wedge t(x_j) \overset{*}{\leftarrow} t(x_k) \right\}$$

- Projection selects all the node identifier columns. Formally,

$$ProjColRefs = x^{cs}_1.id, \ldots, x^{cs}_{n_V}.id$$

- Finally, a renaming is executed. In the result view, the name of each column corresponds to the node from which it originates. Moreover, it stores the identifiers of those objects that were assigned to the original rule graph node by matchings. Note that the result view has as many columns as many nodes its origin rule graph had.

$$ResCols = x_1^d, \ldots, x_{n_V}^d$$

The view for the NACs can be calculated in exactly the same way, but using the NAC graphs in the process. Now we define when a matching is consistent with its database representation.

Definition 52 Given a model M together with a database representation $\mathfrak{M}$, **a matching** m **for a pattern** r_G **in model** M **is consistent with a row** $\vec{m}^d$ **of a view** r_G^d **in database representation** $\mathfrak{M}$ — denoted by $(m|r_G) \cong (\vec{m}^d|r_G^d)$ — (i) if the identifiers of all objects of instance model M that have been selected by matching m for pattern r_G can be found as an element in the corresponding position of row $\vec{m}^d$, and (ii) for each element of a row $\vec{m}^d$ in r_G^d there is a node in pattern r_G that is mapped to the object that corresponds to the given element of the selected row by the matching m. Formally, there exists a matching m for pattern G in model $M \iff \exists \vec{m}^d \in r_G^d, \forall x \in V_G : \vec{m}^d[x^d] = m(x)^d$.

Note that the above definition is asymmetric as pattern matching requires matching model elements both for nodes and edges of the pattern, while the corresponding row in the view contains only the identifiers of matching objects.

Definition 53 Given a model M together with a database representation $\mathfrak{M}$, **a pattern** r_G **is consistent with a view** r_G^d (denoted by $r_G \cong r_G^d$) if (i) for each matching m of a pattern r_G in instance model M there exists a row $\vec{m}^d$ in r_G^d where matching m is consistent with row $\vec{m}^d$ and (ii) for each row $\vec{m}^d$ in r_G^d there exists a matching m of a pattern r_G where matching m is consistent with row $\vec{m}^d$. Formally,

- $\forall m : G \to M, \exists \vec{m}^d \in r_G^d : (m|r_G) \cong (\vec{m}^d|r_G^d)$
- $\forall \vec{m}^d \in r_G^d, \exists m : G \to M : (m|r_G) \cong (\vec{m}^d|r_G^d)$

Finally, a theorem is formulated, which states that each possible matching of a LHS (or NAC) rule graph corresponds to exactly one row in the r_{LHS}^d (or r_{NAC}^d) view. Furthermore, the row in the view contains the identifiers of objects that participate in the matching.

Theorem 2 *Let d be a bidirectional mapping between $\mathfrak{S}_{GT}$ and $\mathfrak{S}_{DB}$. If model M is consistent with the database representation $\mathfrak{M}$, then a pattern r_G (*without *negative application condition) in $\mathfrak{S}_{GT}$ is consistent with view r_G^d in $\mathfrak{S}_{DB}$. Formally, $M \cong \mathfrak{M} \implies r_G \cong r_G^d$.*

PROOF The proof is in Appendix A.

6.4.3 Left joins for preconditions of rules.

As it has been introduced in Sec. 6.2, the calculation of a view for the precondition of a rule proceeds as follows. Each NAC is left outer joined to the LHS graph one by one by using join conditions, which express that columns representing the same shared node in different rule graphs should be equal. Additional filtering conditions require that columns of NAC(s), which are shared with the LHS part, have to be filled with undefined values. Then a *projection* displays only those columns that originate from LHS. Finally, a column renaming procedure performs an identical redefinition of column names.

Formalization. We suppose that rule r consists of a LHS and k negative application conditions. As before, n_V is used for denoting the cardinality of V_{LHS}.

The view generated for the precondition r_{PRE} consists of n_V columns and it can be calculated as follows.

$$r^d_{\mathsf{PRE}}(ResCols) = \pi_{ProjColRefs}\left(\sigma_{Null}(\mathcal{S}_k)\right).$$

- *Left outer join.* Each $r^d_{\mathsf{NAC}_i}$ is left outer joined to r^d_{LHS} one by one using a join condition F_i. Formally, $\mathcal{S}_k = r^d_{\mathsf{LHS}} \overset{F_1}{\ltimes} r^d_{\mathsf{NAC}_1} \overset{F_2}{\ltimes} \ldots \overset{F_k}{\ltimes} r^d_{\mathsf{NAC}_k}$

- *Join conditions.* F_i express that shared nodes cannot be mapped to different objects in the model M by matching functions m of r_{LHS} and m' of r_{NAC_i}. Formally,

$$F_i = \bigwedge \left\{ r^{cs}_{\mathsf{LHS}}.x^d_l = r^{cs}_{\mathsf{NAC}_i}.x^d \mid x_l \in \underline{V_{\mathsf{LHS}}} \cap V_{\mathsf{NAC}_i} \wedge x \in \underline{V_{\mathsf{NAC}_i}} \cap V_{\mathsf{LHS}} \wedge p_{\mathsf{NAC}_i}(x_l) = x \right\}$$

  Note that the fact that column name x^d appearing in several tables only denotes that those columns represent the same (shared) node of the rule graph in tables r^d_{LHS} and $r^d_{\mathsf{NAC}_i}$.

- *Null conditions.* $Null$ express that it is not allowed to have matchings for any r_{NAC_i} in order to have a matching for r_{PRE}. Formally,

$$Null = \bigwedge \left\{ r^{cs}_{\mathsf{NAC}_i}.x^d = \varepsilon \mid i \in \mathbb{Z}^+_k \wedge x \in \underline{V_{\mathsf{NAC}_i}} \cap V_{\mathsf{LHS}} \right\}$$

  In this expression, $\mathbb{Z}^+_k$ denotes positive integers up to k.

- *Projection* selects all columns that originate from view r^d_{LHS}. Formally,

$$ProjColRefs = r^{cs}_{\mathsf{LHS}}.x^d_1, \ldots, r^{cs}_{\mathsf{LHS}}.x^d_{n_V}$$

- Finally, *identical renaming* is implemented. In the result view, the name of each column is the same as the node of the LHS graph from which it originates. Moreover, it stores the identifiers of those objects that were assigned to the LHS graph node by matchings. Note that the result view has as many columns as many nodes r_{LHS} had. Formally,

$$ResCols = x^d_1, \ldots, x^d_{n_V}$$

As a result, each matching for precondition graph r_{PRE} appears as exactly one row in the corresponding view r^d_{PRE}. A row consists of the identifiers of objects that are selected by the matching. In a more formal way, the following theorem can be formulated.

Theorem 3 *Let us suppose that there exists a bijective mapping from $\mathfrak{S}_{GT}$ to $\mathfrak{S}_{DB}$. If model M is consistent with the database representation $\mathfrak{M}$, then a pattern r_{PRE} in $\mathfrak{S}_{GT}$ that* has *negative application conditions is consistent with view r^d_{PRE} in $\mathfrak{S}_{DB}$. Formally, $M \cong \mathfrak{M} \Longrightarrow r_{PRE} \cong r^d_{PRE}$.*

PROOF The proof can be found in Appendix A.

Algorithm 6.2 Edge deletion

Require: $\exists \vec{r} \in r^d \wedge \exists m_r \wedge (m_r|r) \cong (\vec{r}|r^d)$
1: **for all** $u_{del} \overset{z_{del}}{\rightarrow}_1 v_{del} \in E_{\mathsf{LHS}} \setminus E_{\mathsf{RHS}}$ **do**
2: `UPDATE` $src(t(z_{del}))^d$ `SET` $t(m(z_{del}))^d = \varepsilon$ `WHERE` $id = m(u_{del})^d$
3: **end for**
4: **for all** $u_{del} \overset{z_{del}}{\rightarrow}_* v_{del} \in E_{\mathsf{LHS}} \setminus E_{\mathsf{RHS}}$ **do**
5: `DELETE FROM` $t(z_{del})^d$ `WHERE` $src = m(u_{del})^d$ `AND` $trg = m(v_{del})^d$
6: **end for**

6.4.4 Graph manipulation in relational databases

Operations in the graph manipulation phase can be implemented by issuing several data manipulation commands in a single transaction block as it has been explained informally in Sec. 6.2. Note that the database updating algorithm parts should be executed in exactly the same order as it appears in the current section.

Deletions. For each $u_{del} \overset{z_{del}}{\rightarrow} v_{del} \in E_{\mathsf{LHS}} \setminus E_{\mathsf{RHS}}$, the matched edge $m(u_{del}) \overset{m(z_{del})}{\rightarrow} m(v_{del})$ has to be deleted from the model M. In the database the corresponding edge deletion is performed as follows.

- For each many-to-one edge $u_{del} \overset{z_{del}}{\rightarrow}_1 v_{del}$ of the $E_{\mathsf{LHS}} \setminus E_{\mathsf{RHS}}$ set (line 1), an `UPDATE` command should be executed on the table that corresponds to the source node $src(t(z_{del}))$ of the direct type of the edge (line 2).
- For each many-to-many edge $u_{del} \overset{z_{del}}{\rightarrow}_* v_{del}$ of the $E_{\mathsf{LHS}} \setminus E_{\mathsf{RHS}}$ set (line 4), a `DELETE` command should be executed on the table that corresponds to the type of the edge $t(z_{del})$ (line 5).

If $x_{del} \in V_{\mathsf{LHS}} \setminus V_{\mathsf{RHS}}$, then its image $m(x_{del})$ and all the dangling edges (i.e., all incident edges) should be removed from the model M. On the database level even the deletion of a single node is performed by issuing a sequence of DELETE operations. One reason why a single DELETE is insufficient is that a node identifier can appear in several node tables because of inheritance in the metamodel. Moreover, node identifiers may appear in tables that represent edges. These latter types of rows should also be deleted in order to ensure that the instance model still remains a graph.

The node deletion algorithm (see Alg. 6.3) proceeds as follows.

- It iterates through all the nodes of $V_{\mathsf{LHS}} \setminus V_{\mathsf{RHS}}$ (line 1).
- All types of each node belonging to the difference set are determined, and they get ordered according to the inverse topological order (line 2) to prevent violating foreign key constraints during deletion. (The inverse topological order is a bottom-up style enumeration of the ancestors of a specific type.)
- All the outgoing many-to-many associations A_{out} that have class C as their source node have to be determined. (line 3–5)
    - The appropriate DELETE command can be executed on the tables that correspond to the above-mentioned association. (line 4)
- All the incoming many-to-many associations A_{in} that have class C as their target node have to be determined. (line 6–8)

Algorithm 6.3 Node and dangling edge deletion

Require: $\exists \vec{r} \in r^d \wedge \exists m_r \wedge (m_r|r) \cong (\vec{r}|r^d)$

1: **for all** $x_{del} \in V_{\mathsf{LHS}} \setminus V_{\mathsf{RHS}}$ **do**
2: **for all** $C \in InverseTopologicalOrder(t(m(x_{del})))$ {List ancestors of $t(m(x_{del}))$ in a bottom-up order} **do**
3: **for all** $C \overset{A_{out}}{\rightarrow}_* D_1 \in Assoc_{M2M}$ {For all outgoing many-to-many associations A_{out} having source class C} **do**
4: `DELETE FROM` A^d_{out} `WHERE` $src = m(x_{del})^d$
5: **end for**
6: **for all** $D_2 \overset{A_{in}}{\rightarrow}_* C \in Assoc_{M2O}$ {For all incoming many-to-many associations A_{in} having target class C} **do**
7: `DELETE FROM` A^d_{in} `WHERE` $trg = m(x_{del})^d$
8: **end for**
9: **for all** $D_3 \overset{A_{in}}{\rightarrow}_1 C \in Assoc_{M2M}$ {For all incoming many-to-one associations A_{in} having target class C} **do**
10: `UPDATE` D^d_3 `SET` $A^d_{in} = \varepsilon$ `WHERE` $A^d_{in} = m(x_{del})^d$
11: **end for**
12: `DELETE FROM` C^d `WHERE` $id = m(x_{del})^d$ {Deletes the object itself from C^d and all outgoing many-to-one links, which have been stored in C^d}
13: **end for**
14: **end for**

- A similar DELETE command has to be executed on the tables that correspond to the above-mentioned association. (line 7)

- All the incoming many-to-one associations A_{in} that have class C as their target node have to be determined. (line 9–11)
  - An UPDATE command has to be executed on the tables that correspond to the source nodes of the above-mentioned associations. (line 10)

- Finally, the node itself can be deleted from class C (line 12), and the iteration should be continued on the ancestors of C. Note that this step automatically deletes all outgoing many-to-one links, which have been stored in table C^d.

Insertions. If a node x_{ins} appears only in RHS, but not in LHS, then a new object (denoted by $m_{\mathsf{RHS}}(x_{ins})$) of type $t(x_{ins})$ should be added to model M.

- The algorithm iterates over each node x_{ins} that appears only in RHS, but not in LHS (line 1–6).
- A new identifier $m_{\mathsf{RHS}}(x_{ins})^d$ is generated (line 2).
- On each ancestor of $t(x_{ins})$ (line 3–5) an `INSERT` operation is executed (line 4).

If $u_{ins} \overset{z_{ins}}{\rightarrow} v_{ins} \in E_{\mathsf{RHS}} \setminus E_{\mathsf{LHS}}$, then a new link $m_{\mathsf{RHS}}\left(u_{ins} \overset{z_{ins}}{\rightarrow} v_{ins}\right)$ of type $t(z_{ins})$ should be added to the model M.

Algorithm 6.4 Node insertion

Require: $\exists \vec{r} \in r^d \wedge \exists m_r \wedge (m_r|r) \cong (\vec{r}|r^d)$

1: **for all** $x_{ins} \in V_{\mathsf{RHS}} \setminus V_{\mathsf{LHS}}$ **do**
2: $m_{\mathsf{RHS}}(x_{ins})^d := GenerateNewIdentifier()$ {Generates identifier for the new node}
3: **for all** $C \in TopologicalOrder(t(x_{ins}))$ {Top-down traversal of class hierarchy ending in $t(x_{ins})$} **do**
4: `INSERT INTO` C^d (id) `VALUES` $(m_{\mathsf{RHS}}(x_{ins})^d)$ {Inserts the new object identifier into column id, which stores identifiers of objects of type C}
5: **end for**
6: **end for**

Algorithm 6.5 Edge insertion

Require: $\exists \vec{r} \in r^d \wedge \exists m_r \wedge (m_r|r) \cong (\vec{r}|r^d)$

1: **for all** $u_{ins} \overset{z_{ins}}{\rightarrow}_1 v_{ins} \in E_{\mathsf{RHS}} \setminus E_{\mathsf{LHS}}$ **do**
2: `UPDATE` $src(t(z_{ins}))^d$ `SET` $t(z_{ins})^d = m_{\mathsf{RHS}}(v_{ins})^d$ `WHERE` $id = m_{\mathsf{RHS}}(u_{ins})^d$
3: **end for**
4: **for all** $u_{ins} \overset{z_{ins}}{\rightarrow}_* v_{ins} \in E_{\mathsf{RHS}} \setminus E_{\mathsf{LHS}}$ **do**
5: `INSERT INTO` $t(z_{ins})^d$ (src, trg) `VALUES` $(m_{\mathsf{RHS}}(u_{ins})^d, m_{\mathsf{RHS}}(v_{ins})^d)$
6: **end for**

- For each many-to-one edge $u_{ins} \overset{z_{ins}}{\rightarrow}_1 v_{ins}$ that can be found in $E_{\mathsf{RHS}} \setminus E_{\mathsf{LHS}}$ (line 1–3), an `UPDATE` command should be executed on the table that corresponds to the source node $src(t(z_{ins}))$ of the direct type of the edge (line 2).
- For each many-to-many edge $u_{ins} \overset{z_{ins}}{\rightarrow}_* v_{ins}$ of $E_{\mathsf{RHS}} \setminus E_{\mathsf{LHS}}$ (line 4–6), an `INSERT` command should be executed on the table that corresponds to the type of the edge $t(z_{ins})$ (line 5).

Now we can formulate the final statement that expresses the correct behaviour of our algorithm. This states that if a model M was consistent with its database representation $\mathfrak{M}$, and if we perform modifications on the model by a graph transformation rule and we execute the corresponding updating algorithm in the database, then the resulting model M' and the database representation $\mathfrak{M}'$ will still be consistent, yielding that our algorithm built on top of a relational database correctly performs graph transformation.

Theorem 4 *Let us suppose that there exists a bijective mapping d from $\mathfrak{S}_{GT}$ to $\mathfrak{S}_{DB}$. If (i) model M is consistent with the database representation $\mathfrak{M}$, (ii) we have a matching m_r for rule r, together with a corresponding row $\vec{m}^d$ in view r^d, and m is consistent with $\vec{m}^d$, (iii) rule r is applied on matching m_r resulting in M', and (iv) Algorithms 6.2–6.5 are executed in the database for $\vec{m}^d \in r^d$ resulting in a database representation $\mathfrak{M}'$, then $M' \cong \mathfrak{M}'$.*

Formally, if

(i) $M \cong \mathfrak{M}$,
(ii) $(m_r|r) \cong (\vec{m}^d|r^d)$ *for a pair* $(m_r, \vec{m}^d)$,
(iii) $M \overset{r,m_r}{\Longrightarrow} M'$,
(iv) $\mathfrak{M} \overset{Alg.\,6.2-6.5}{\Longrightarrow} \mathfrak{M}'$,

then $M' \cong \mathfrak{M}'$.

PROOF The proof can be found in Appendix A.

6.5 Measurement results

The quantitative performance analysis of RDBMS based graph transformation already started in Sec. 5.5, where the approach was compared to other tools. In the current experiments, we focus on such properties of our approach that are expected to have a significant impact on run-time performance or that are specific to our database related solution. The performance measurements of this chapter have been executed on the object-relational mapping benchmark example, which has already been introduced in Sec. 5.4.

According to the performance analysis of Sec. 5.5, the most significant speed-up could be observed in case of a database related approach when *parallel rule execution* is used as an optimization strategy. As a consequence, only this tool feature is included into the experiments of the current chapter.

An additional optimization possibility is identified, which is specific to a graph transformation approach that is based on top of a relational database. This database specific tool feature is the *application of the built-in query optimizer* of the underlying RDBMS. Note that the query built for the precondition of a graph transformation rule has a special structure, for which the built-in query plan generator may not provide an optimal solution as it lacks the additional information about the structure of GT rules or models. Since some relational databases allow the definition of such queries, for which the generated plan can be influenced from outside the RDBMS, the examination of this optimization possibility has been included in the measurements. The queries prepared for the own optimization strategy were made by hand and they were based on the same application domain dependent engineering guidelines that are used in many graph transformation tools.

As two orthogonal tool features have been identified, the measurements were performed on all the four possible combinations of these features, which means that four test cases have been analyzed. The runtime parameter N, which denotes the maximum number of processes during the run, was fixed to 10 and 30 in test cases where rules were executed sequentially, and N was set to 10, 30, 50 and 100 for test cases with parallel rule application.

Two popular RDBMSs (namely MySQL version 4.1.7 and PostgreSQL version 8.0.3) took part in our measurements, which were performed on a 1500 MHz Pentium machine with 768 MB RAM. A Linux kernel of version 2.6.7 served as an underlying operating system. The execution time results are shown in Table 6.1.

The head of a row (i.e., the first two columns) shows the name of the rule and the optimization strategy settings for the single tool feature (i.e., parallel rule execution) on which the average is calculated. (Note that a rule is executed several times in a run.) The third column (Class) depicts the number of classes in the run, which is, in turn, the runtime parameter N of the test case. The fourth and fifth columns show the concrete values for the model size and the transformation sequence length, respectively. Heads of the remaining columns unambiguously identify the RDBMS used and the status denoting whether the built-in query optimizer was used (db) or not (own). Values in match and update columns depict the average times needed for a single execution of a rule in the pattern matching and updating phase, respectively. Execution times were measured on a microsecond scale, but a millisecond scale is used in Table 6.1 for presentation purposes. Light grey areas denote run-time failures due to exceeding the default memory allocation limits of the operating system.

Our experiments can be summarized as follows.

	ObjRel	Class	Model size	TS length	MySQL				PostgreSQL			
					db		own		db		own	
		#	#	#	match msec	update msec	match msec	update msec	match msec	update msec	match msec	update msec
AssocEndRule	parallel OFF	10	1342	146	24.23	2.91	29.45	3.50	27.63	4.40	53.40	4.46
		30	12422	1336	543.41	2.74	549.97	2.73	127.22	6.39	679.81	5.15
	parallel ON	10	1342	146	0.23	3.28	0.23	3.39	2.60	6.23	1.00	4.07
		30	12422	1336	0.13	2.83	0.40	2.40	0.40	5.97	0.80	6.14
		50	34702	3726	0.37	3.93	0.14	5.22	0.26	4.77	1.53	5.34
		100	139402	14951	0.12	4.24	0.12	4.68	0.58	7.69		
AssociationRule	parallel OFF	10	1342	146	12.20	4.82	13.60	5.18	5.57	5.60	4.29	6.72
		30	12422	1336	160.20	2.94	159.41	2.96	37.20	4.90	48.62	5.62
	parallel ON	10	1342	146	0.38	4.43	0.26	6.13	0.22	6.05	0.26	5.61
		30	12422	1336	0.12	2.91	0.11	2.98	0.08	5.90	0.09	3.77
		50	34702	3726	0.10	2.71	0.10	3.24	0.08	8.19	0.08	8.03
		100	139402	14951	0.08	4.43	0.07	4.88	0.06	6.39		
ClassRule	parallel OFF	10	1342	146	13.17	2.68	14.28	3.14	7.29	5.31	5.86	5.41
		30	12422	1336	249.38	3.04	247.82	2.68	32.95	5.08	32.91	5.01
	parallel ON	10	1342	146	1.33	2.94	1.35	2.94	0.82	4.81	0.81	4.86
		30	12422	1336	7.41	2.38	7.44	2.35	1.25	4.07	1.09	4.12
		50	34702	3726	39.78	1.99	38.32	2.04	1.99	3.80	2.00	3.74
		100	139402	14951	262.40	2.00	268.99	1.95	8.37	3.62		

Table 6.1: Experimental results

- In accordance with our assumptions, parallel rule execution has a dramatic effect on pattern matching. The time increase for ClassRule can be explained by having a constant initialization and resource allocation time, which is distributed over a relatively small number of rule applications.
- We have been forced to use temporary tables instead of views in case of MySQL version 4.1.7 as it does not support the concept of views. This obligate choice has a strong negative impact in case of sequential rule execution on the performance of the graph transformation engine as temporary tables are always stored on disks in contrast to views (of PostgreSQL), which are calculated in the memory in general.
- The update phase is slightly longer for PostgreSQL, but the difference cannot be considered significant as the execution times for both databases are of the same order of magnitude.
- The results for query plans own being generated and injected by the GT engine may deviate in both directions from the results of plans db that have been created by the query optimizer. This observation indicates that it is possible to create queries with better performance than the ones that are produced by RDBMS, which is an argument for doing further research on generating special queries optimized for GT rules.
- In contrast to our assumptions, MySQL does not allow manual influence on query plan generation, which is indicated by the similar values in its db and own columns.
- Since the presented values are calculated as the average of the execution times measured while applying the same rule for several times, Table 6.1 is inappropriate for assessing the exact distri-

bution of measured values. However, as a general observation, it may be stated that the updating phase can typically be characterized by a balanced distribution, in which all the measured values (including the extremes) can be covered by a 3 millisecond interval even in case of the largest models. On the other hand, when pattern matching is performed on a relational database, all the matchings are already available when the first matching is requested. As a consequence, the first matching is calculated several orders of magnitude slower than all the consecutive matchings, which could always be enumerated in 10 microseconds.

6.6 Graph transformation with portable EJB QL queries

As mentioned in Sec. 2.3.1, EJB3-based enterprise applications use the underlying relational database of the application server to store business data. By using the DB approach already presented in this chapter, business functionalities specified by graph transformation can also be performed by using SQL queries. However, this solution would not be portable in an enterprise environment as the underlying relational databases typically use different dialects of SQL.

In this section, we propose a database independent and portable GT engine implementation, which uses the declarative queries of the Enterprise Java Beans Query Language (EJB QL) [130] in the graph pattern matching phase instead of SQL commands, which aims at resolving the diversity of SQL dialects. Since this new proposal is highly similar to the DB approach in its concepts, only a less detailed presentation of the graph pattern matching phase is given.

Basic concepts of Java based enterprise applications and their runtime execution environment have been introduced in Sec. 2.3 together with the representation of metamodels and models. Now the basics of the declarative query specification language of EJB3 is discussed, which is followed by the presentation of the EJB QL based pattern matching approach.

6.6.1 Enterprise Java Beans Query Language

An application server has an entity manager unit, which provides operations for creating and removing persistent entity instances, for finding entities by their primary key, and for querying over entities.

Queries can be specified in the declarative, object-oriented EJB Query Language (EJB QL) [130]. Only the structure of the SELECT statement is presented here in Listing 6.1 as it is the only construct that is used in the upcoming section.

```
SELECT select_clause
FROM from_clause
WHERE where_clause
```

Listing 6.1: General structure of the SELECT statement in an EJB QL query

The SELECT *clause* denotes the result of the query by a comma separated list of identification variables. An *identification variable* is a variable that can refer to a single instance of a particular entity bean class.

The FROM *clause* designates the domain of the whole SELECT statement by a comma separated list of *identification variable declarations* of the form $type$ AS new_var. The $type$ of an identification variable new_var can be defined explicitly by using the name of an entity bean class, or implicitly by navigating along links of type assoc from an already declared variable old_var. In the latter case, the target class of assoc defines the type of identification variable new_var. Navigation is defined by

```
SELECT pattern_node_variables
FROM identification_variable_declarations
WHERE initial_matching_constraints AND type_checking_constraints
  AND check_edge_constraints AND injectivity_constraints AND NAC_constraints
```

Listing 6.2: EJB QL query representing the pattern matching phase

path expressions *old_var*.assoc or IN(*old_var*.assoc), if the target end has an at most one or arbitrary multiplicity constraint, respectively.

The optional WHERE *clause* is a Boolean expression, and it filters out those results of the query that do not satisfy this expression. A *Boolean expression* is the conjunction (logical AND) of Boolean valued factors, which may test the non-existence of results for a well-formed subquery (NOT EXISTS (*subquery*)), the equality of simple factors (sf_1=sf_2), and the inequality of simple factors (sf_1<>sf_2). A *simple factor* can be a constant, or a navigation operation (denoted by *var*.id) to access the identifier id of an identification variable *var*.

6.6.2 Graph pattern matching on EJB3 platform

By using search plans of LHS and embedded NAC patterns, we construct and execute a single SELECT EJB QL query that calculates and retrieves all the matchings of the precondition of a rule.

The general form of the query is as follows:

A *pattern node variable* is an identification variable being declared in the FROM clause of the EJB QL query, which represents a pattern node. Identification variables that represent bound and free nodes are called *bound node variables* and *free node variables*, respectively.

The SELECT clause of the query contains all pattern node variables in the form of a comma separated list.

In order to handle the mappings of (initially matched) bound nodes, an identification variable declaration of the form $type_{bound}$ AS *bound* is appended to the FROM clause. Additionally, an expression of the form *bound*.id = *obj* is added as an *initial matching constraint*, in which *obj* denotes the object to which bound node *bound* has been mapped by the initial matching.

If a free (pattern node) variable is reached by navigation in the FROM clause of an EJB QL query, then the type of this pattern node variable may be an ancestor of the type prescribed by the pattern node itself. This yields a situation where the pattern node variable has a possibly larger set of matching objects than it is allowed by the type restriction set up by the pattern node. In order to resolve this situation, an additional identification variable is declared for representing the same pattern node and a *type checking constraint* is defined to narrow the set of matching objects for this pattern node.

Free node variable declarations and type checking constraints are generated during search plan traversal while processing search plan edges in an increasing order.

- **Processing iteration edges.** If an iteration edge with a target node *trg* is being processed, then an expression $type_{trg}$ AS *trg* is added to the end of the FROM clause where $type_{trg}$ is the type of the pattern node *trg*.

- **Processing to-one navigation edges.** If a to-one navigation edge of type assoc connecting node *src* to *trg* is being processed, then expressions *src*.assoc AS *trg*_sup and $type_{trg}$ AS *trg* are appended to the end of the FROM clause, and a subformula *trg*_sup.id = *trg*.id is also added as a type checking constraint.

- **Processing to-many navigation edges.** If a to-many navigation edge of type assoc connecting node src to trg is being processed, then terms `IN(`src`.assoc) AS` trg`_sup`, and $type_{trg}$ `AS` trg are appended to the FROM clause, and a subformula trg`_sup.id =` trg`.id` is also added as a type checking constraint.

An *edge checking constraint* expresses a restriction, which is caused by a pattern edge that has not been processed at all during the traversal of the search plan. For each pair of unnumbered navigation edges connecting nodes src and trg in both directions, we append a subformula src`.assoc =` trg`.id` or trg `MEMBER OF` src`.assoc` to the WHERE condition by using a logical AND operator for affixing, if src.`assoc` represents a to-one or a to-many navigation edge, respectively.

Injectivity constraints are defined for such pairs of pattern nodes where one member has a type that conforms to a supertype of the other. For each such pair $node_i$ and $node_j$, we add a subformula of the form $node_i$`.id <>` $node_j$`.id`.

NAC constraints express restrictions formulated by NAC patterns that are embedded into the LHS pattern being processed. For each embedded NAC pattern, we add a constraint of the form `NOT EXISTS (`$subquery$`)`, where $subquery$ is the EJB QL query that is going to be generated for the embedded NAC pattern. In case of NAC subqueries, bound node variables do not have to be declared in the FROM clause, as their declarations are already contained by the embedding query of the LHS. The `NOT EXISTS` constraint will be evaluated to true if and only if the subquery, which would list the matchings of the NAC pattern has no rows.

Example 20 To continue our running example, we present the SELECT statement (Listing 6.3) that is generated for the search plans of the LHS and NAC patterns of ClassRule (as depicted in the left and right parts of Fig. 6.3, respectively). Note that in case of search plans prepared for EJB QL queries, NAC nodes are always marked by the maximum possible value as the subqueries of NAC patterns are always evaluated after matching the LHS pattern.

Figure 6.3: Sample search plans of the LHS and the NAC of ClassRule

Lines 1–9 of the query are generated during the traversal of the search plan of LHS, when its edges are processed in increasing order as shown by the comments at the ends of lines. (Expressions in parentheses denote the search plan edge processing method being used.) Since at least one edge is selected from each pair of navigation edges, no edge checking constraints are needed in the query. Metamodel class Schema is a subclass of class Package, so schema S cannot be mapped to the same object as package P as expressed by Lines 8–9. The query for the NAC pattern (Lines 10–16) is processed similarly with the single exception that C now counts as a bound node as a mapping for node C has already been found.

```
SELECT s,p,c
FROM Schema AS s,                        -- 1 (iter)
   s.ref AS p_sup, Package AS p,         -- 2 (one)
   IN(p.eo) AS c_sup, Class AS c         -- 3 (many)
WHERE -- type checking constraints
   p_sup.id = p.id AND c_sup.id = c.id
   -- no edge checking constraints
   -- injectivity constraints
   AND s.id <> p.id
   -- NAC constraint
   AND NOT EXISTS (
      SELECT c,tn
      FROM c.ref AS tn_sup, Table AS tn -- 1 (one)
      WHERE tn_sup.id = tn.id
      AND c.id <> tn.id
   )
```

Listing 6.3: The EJB QL query describing the pattern matching defined by search plans of Fig. 6.3

6.7 Conclusion

In the current chapter, I elaborated a provenly correct method for executing graph transformation built on top of a relational database, and assessed the performance of the approach in several measurement settings involving different combinations of databases, parameters and optimization strategies.

- *Graph pattern matching in relational databases.* I elaborated a provenly correct method, which automatically transforms graph patterns to SQL queries whose behaviour corresponds to the pattern matching phase of graph transformation (Sections 6.4.2 and 6.4.3).
- *Modification phase of graph transformation on top of relational databases.* I elaborated a provenly correct method, which generates such SQL commands, whose behaviour corresponds to the updating phase of graph transformation (Sec. 6.4.4).
- *Quantitative performance analysis of the method.* By using the object-relational mapping as a benchmark example, I examined the efficiency of this technique in several measurement settings involving different combinations of databases, parameters and optimization strategies (Sec. 6.5).
- *Portable queries for graph pattern matching.* I proposed a database independent and portable pattern matching approach that uses declarative EJB QL queries to implement graph pattern matching (Sec. 6.6).

These results are reported in [145, 149, 150, 151].

Relevance

The relevance of the presented approach can be summarized by examining the motivation goals laid out in Sec. 6.1.

As the RDBMS based graph transformation approach stores models on the disk, it has the ability to handle larger models compared to pure in-memory GT engines. Performance experiments of Sections 5.5 and 6.5 demonstrate that relational databases provide a feasible candidate as an implementation framework for graph transformation engines with promising results especially for large models.

However, performance is not the only aspect one needs to consider from a practical point of view when implementing model transformations. Our relational database approach automatically provides persistence and transactional services without further programming effort.

Persistence is very important in the case of MDA tools storing their UML models in relational databases as e.g., AMEOS of Aonix [4]. This tool offers powerful built-in means to capture model-to-code transformations, but model-to-model transformations (including model manipulations) are not supported, which could be complemented by our technique to provide a general solution.

When model transformation is used in an enterprise environment, transformation plugins are typically implemented as session beans whose methods are generally executed in a transaction block, which should provide support for withdrawing the effects of rule execution. As RDBMSs are able to handle transactions by default, our approach can easily integrated into an enterprise environment to function as a model transformation plugin, in which pattern matching is expressed by (SQL or EJB QL) queries.

A further advantage of the presented technique is that it is also suitable for in-memory RDBMSs like TimesTen [106], SQLite [124], Xcelerix [160], HSQLDB [65] without modification. In such an approach, performance and model size are expected to be on the same order of magnitude as in case of any other in-memory solutions.

The practical applicability of RDBMS based graph transformation has recently been confirmed in [71] by the developers of the MOLA tool [69, 70], who used a similar approach for implementing their Transformation Execution Environment module.

Limitations

The presented approach suffers from certain runtime overhead due to several reasons. The underlying database performs statement parsing, optimization, table (or row) locking, and transaction handling tasks during the execution of each query. Though repeated parsing can be partially avoided by prepared statements, and both optimization and transaction handling can sometimes be guided by the pattern matcher, each remaining task obviously increases the execution time of the pattern matching engine.

The application server also introduces performance degrading factors in case of EJB3-based pattern matching solutions as the objects in the matchings returned are also cached in the main memory in addition to their original location on the disk, and the object-relational mapping service of the application server has to synchronize these copies. Additionally, the application server usually provides transaction handling services whose usage also worsens performance.

As a natural limitation of the EJB QL based approach, it is worth emphasizing the trade-off between portability and run-time performance when database-specific query optimizations are switched on and off.

CHAPTER

7 Adaptive Graph Transformation

In this chapter, I present a novel approach to implement adaptive, and model-sensitive graph pattern matching modules. Based on the statistics of the instance model under transformation, these modules can dynamically select the optimal pattern matching strategy from a set of precompiled strategies that have been generated from model-sensitive search plans by using estimates for their expected performance on typical instance models available at transformation design time. As a result, a fast transformation engine can be obtained, which can dynamically modify its behaviour at run-time in the performance critical pattern matching phase by always selecting the strategy that is expected to be optimal for the instance model under transformation.

7.1 Motivation

As model transformation is becoming an engineering discipline (*transware*), 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, *interpreted MT approaches* have a clear advantage during the validation (e.g., by interactive simulation) or the maintenance phase due to their flexibility. On the other hand, the main driver in the execution phase is performance, therefore, a *compiled MT approach* is advantageous as shown by survey [154].

The performance of the executable code is optimized at compile time by evaluating and optimizing different *search plans* [163] for the traversal of the LHS pattern, which typically *exploits the multiplicity and type restrictions* imposed by the metamodel of the problem domain.

Problem statement

- **Lack of adaptivity.** While in many cases, types and multiplicities provide a very 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.
- **Hard-wired pattern matching strategies.** Furthermore, in case of intensive changes during the evolution of models, the characteristic structure of a model may change as well, therefore a

pattern matching strategy that is generated a priori at compile time from a search plan might not be flexible and powerful enough.

Objectives

I propose a method for generating model-sensitive search plans for 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. I also elaborate a method for finding low cost search plans with respect to the cost function that estimates the performance of a given strategy.

Furthermore, I propose an adaptive approach for graph pattern matching, where the optimal strategy can be selected from previously generated pattern matching strategies at run-time based on statistical data collected from the instance model under transformation.

Finally, I present a technique for preparing adaptive, compiled, stand-alone plugins for representing pattern matching strategies on the EJB 3.0 platform from model-sensitive search plans.

Related work

Sophisticated pattern matching strategies of the most advanced compiled graph transformation approaches are now surveyed.

- FUJABA [99] compiles visual specifications of transformations [44] 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.

- PROGRES [122] supports both interpreted and compiled execution (generating C code) of programmed graph transformation systems. It uses a sophisticated cost model [163] for defining a priori costs of basic operations (like the enumeration of nodes of a type and navigation along edges) for generating search plans. 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 of Sec. 4.2, 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 compiled GREAT [158] (generating C++ code) 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 [126] 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.

In the graph transformation community, adaptivity has already been used with a completely different meaning, when adaptive star grammars [33] have been defined for applications, which are used

for modeling and refactoring object-oriented programs. In this context, an adaptive star grammar is an extension of graph grammars, and it consists of rule schemas, each of which can describe a potentially infinite number of concrete rules.

Structure of the chapter

The proposed workflow of building an adaptive graph transformation engine is summarized in Fig. 7.1.

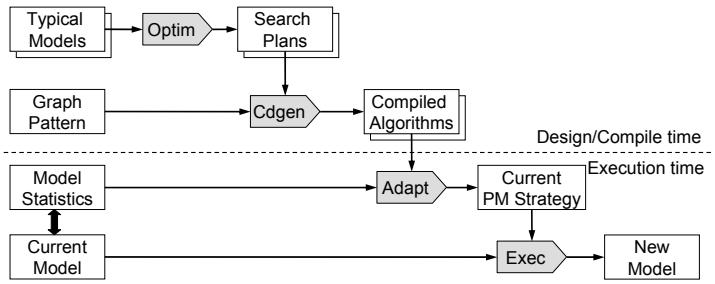


Figure 7.1: Overview of the approach

- Section 7.2 extends model repositories by introducing statistical data collection from the instance model under transformation in order to support model-specific concepts.
- Section 7.3 presents the optimization task (Optim), in which 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 at transformation design time.
- Section 7.4 describes the code generation task (Cdgen), which produces adaptivity enabled, compiled pattern matching strategies by generating methods for cost calculation beyond pattern matching code fragments to support run-time performance estimation for these strategies without their actual execution.
- Section 7.5 illustrates the behaviour of the adaptive engine at execution time. In the adaptation phase Adapt, statistical data is collected on-the-fly from the current model under transformation. Based on this data, a pattern matching strategy (i.e., an implementation of a search plan) is selected which yields the best expected performance cost. Finally, in the execution phase Exec, the transformation rule is applied on the instance model using the selected pattern matching strategy.
- Section 7.6 examines and compares the run-time efficiency of Java, EJB3 and EJB QL based implementations by using a fixed search plan as the pattern matching strategy and the object-relational mapping as the benchmark example.
- Section 7.7 concludes this chapter and summarizes its relevance.

7.2 Collecting model statistics

Model-specific concepts require the model repository to support the maintenance of statistical data collected from the model under transformation. For this reason, counters are introduced, whose role is to count the number of objects (links) in the model that can be matched to a given pattern node

(edge). An *object counter* for a class maintains the total number of type conformant objects (denoted by #(Class)) appearing in the model. A *link counter* for an association stores the number of links that conform to the association, and that are further constrained by two additional classes, which prescribe type conformance to their source and target objects, respectively. Link counters are denoted by #(Association,SourceClass,TargetClass) in examples.

This model statistics will be heavily used both for the search plan optimization and the search plan adaptation steps. Note that the overhead caused by the maintenance of these counters is relatively cheap: one option is to use class-level (static) attributes and methods. In fact, in many cases, such details are already provided by the execution environment (e.g., by a relational DBMS, if models are persisted in a database).

The counters are declared in the repository according to the following rules.

- An object counter is added for the direct type of each pattern node.
- A link counter is added to the repository for the direct type of each pattern edge. This link counter is restricted by the direct type of source and target nodes of the corresponding pattern edge.

Example 21 The object-relational mapping has been selected as a running example for the current chapter as well. Its metamodel and the set of graph transformation rules have already been presented in Figures 2.1 and 3.1. Concepts of the current chapter are going to be illustrated on the AssocEndRule, which is shown in Fig. 3.1(f) and also repeatedly in Fig. 7.2(a). A set of sample models has been defined by the corresponding test set in Fig. 5.8. For presentation purposes, models of Figures 5.8(c) and 5.8(e) are repeated in Figures 7.3(a) and 7.3(b), respectively.

Since AssocEndRule contains a class C in its LHS, an object counter for Class is declared. This counter stores the number of classes in the model, which is 4 in case of the model of Fig. 7.3(a). Note that association a12 and table t3 should also be considered as classes due to the fact that Association and Table are inherited from Class as specified by the metamodel of Fig. 2.1.

As the LHS of AssocEndRule has an edge r1 of type Ref connecting class C to table Tc, a link counter #(Ref,Class,Table) is added for association Ref. The set of counted links of type Ref is restricted by requiring their source and target objects to conform to classes Class and Table, respectively. Though the model of Fig. 7.3(a) contains 4 links of type Ref, only two of them (namely the link connecting association a12 to table t3 and the loop of table t3) fulfill the source and target class restrictions, thus, #(Ref,Class,Table) = 2.

Now the formal definitions of counters are presented.

Definition 54 Given a metamodel MM, and a model M, an **object counter for a class** C (denoted by $\#_C$) is a function that maps model M to the cardinality of the set consisting of such objects c in model M whose direct type $t(c)$ is a descendant of class C. Formally,

$$\#_C(M) = \left| \left\{ c \mid c \in V_M \wedge C \overset{*}{\leftarrow} t(c) \right\} \right|.$$

Definition 55 Given a metamodel MM, a model M, and classes S and T that are descendants of the source and target classes of association $C_s \overset{A}{\rightarrow} C_t$, respectively, a **link counter for an association** $C_s \overset{A}{\rightarrow} C_t$ **restricted by classes** S **and** T (denoted by $\#_{(A,S,T)}$) is a function that maps model M to the cardinality of the set consisting of such links $a \overset{e}{\rightarrow} b$ of type A in model M whose source and target objects conform to classes S and T, respectively. Formally,

$$\#_{(A,S,T)}(M) = \left| \left\{ a \overset{e}{\rightarrow} b \mid a \overset{e}{\rightarrow} b \in E_M \wedge t(e) = A \wedge C_s \overset{*}{\leftarrow} S \overset{*}{\leftarrow} t(a) \wedge C_t \overset{*}{\leftarrow} T \overset{*}{\leftarrow} t(b) \right\} \right|.$$

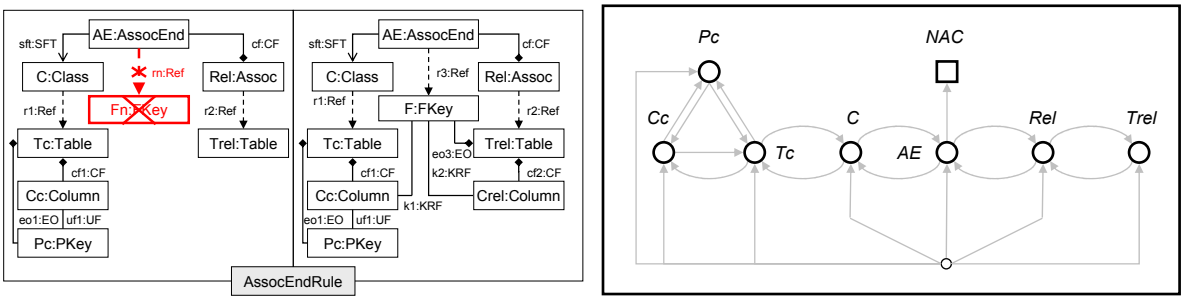
7.3 Generating model-specific search plans

Section 4.2 already introduced the overall approach of search plan driven pattern matching including the constructs of search graphs and plans, and the pattern matching strategy generation process. In the current section, this concept is extended by making search graphs and plans model-sensitive and by introducing a cost model for search plans in order to generate better pattern matching strategies for models that have been selected by the system designer as representatives of a transformation sequence (Sec. 7.3.1). In addition, two algorithms are presented in Sec. 7.3.2 for generating low cost search plans from model-specific search graphs.

7.3.1 Model-specific search graphs and plans

In the first phase of the search plan generation process, a search graph is created for the LHS and NAC patterns of each rule in the same way as described in Sec. 4.2.1.

Example 22 Graph transformation rule AssocEndRule and its corresponding search graph are depicted in Figures 7.2(a) and 7.2(b), respectively.



(a) AssocEndRule (b) Search graph for AssocEndRule

Figure 7.2: A sample graph transformation rule and its corresponding search graph

At this point, the transformation designer selects 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 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 Figures 7.3(c) and 7.3(d).) Informally, the weight of an edge can be considered as an average branching factor of a possible search space tree at the level, when the given pattern edge is selected for navigation. Such a choice for edge weights provides an easy to calculate cost function that estimates the size of the search space.

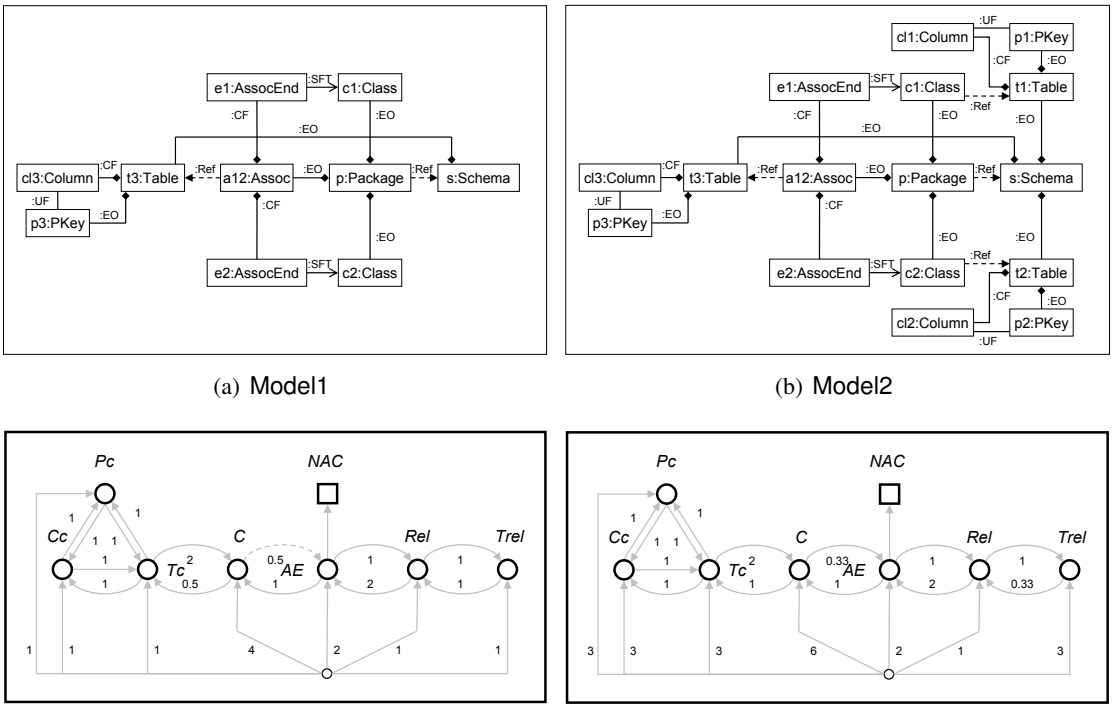
Weight calculation rules can be summarized as follows.

- The weight of an iteration edge corresponds to the number of objects that conform to the type of the pattern node that is represented by the target node derivative of the iteration edge. This value is given by the object counter declared for the type of the target node derivative.
- In case of a navigation edge, first, the number of such links has to be determined that conform to all type restrictions prescribed by the pattern edge (i.e., constraints on the type of the link,

the source object and the target object) that represents the navigation edge in turn. This value is shown by a corresponding link counter, which is restricted by the type constraints of the pattern edge. Since an average branching factor is aimed to be calculated for the weight, the value of the link counter has to be divided by the number of objects that conform to the type of the source node of the pattern edge, which is given by the object counter declared for this type.

Since the dynamic positioning of NAC checking operations requires more sophisticated techniques (as pointed out by [64]), these operations are currently handled in a static way by always appending them to the end of search plans. In this sense, they are also excluded from the model-sensitive search plan generation process, which is reflected by the missing weights on NAC check edges in the weighted search graphs. In the future, complex search plan operations (such as NAC checking and the invocation of recursive patterns [152]) are also aimed to be handled adaptively, and they are planned to be integrated into the general framework of [64].

Example 23 Two models and their corresponding weighted search graphs are depicted in Fig. 7.3.



(a) Model1

(b) Model2

(c) Weighted search graph of the LHS pattern of GT rule AssocEndRule based on the statistics of Model1

(d) Weighted search graph of the LHS pattern of GT rule AssocEndRule based on the statistics of Model2

Figure 7.3: Sample instance models and corresponding weighted search graphs

The weight calculation rule is demonstrated on the navigation edge of Fig. 7.3(c) connecting free node C to AE (denoted temporarily by a dashed line), which corresponds to the reverse traversal of pattern edge sft of type SFT (i.e., from pattern node C to pattern node AE).

According to our statistics, Model1 contains 4 classes including classes c1 and c2, association a12, and table t3. (Note that tables and associations should also be considered as classes according to the corresponding metamodel of Fig. 2.1.) Additionally, Model1 has 2 SFT links between association ends and classes. As a consequence, if the pattern matching engine matches a Class to the pattern node C at some

time during the execution, then the probability to find a valid AssocEnd for pattern node AE by navigating along an SFT edge is 0.5 derived by the formula #(SFT,AssocEnd,Class) / #(Class). In case of navigation in the opposite direction, the formula can be expressed as #(SFT,AssocEnd,Class) / #(AssocEnd), thus the corresponding weight is 1.

Definition 56 Given a metamodel MM and a graph transformation rule r, the **weighted search graph** SG^M **of the** LHS **pattern based on the statistics of model** M is the search graph SG of the LHS pattern together with a model-dependent weight function $w_M : E_{\mathsf{SG}} \to \mathbb{R}^+$ that assigns non-negative numbers to the edges of the search graph according to the following rules.

- The weight of an iteration edge $\mathsf{d} \xrightarrow{\mathsf{i}} \mathsf{x}$ connecting the dummy node d to pattern node derivative x is the value of the object counter that has been declared for the direct type $t(b(\mathsf{x}))$ of the origin pattern node $b(\mathsf{x})$. Formally, $\forall \mathsf{d} \xrightarrow{\mathsf{i}} \mathsf{x} \in E_{\mathsf{SG}} : w_M(\mathsf{d} \xrightarrow{\mathsf{i}} \mathsf{x}) = \#_{t(b(\mathsf{x}))}(M)$.
- The weight of a navigation edge $\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v}$ connecting pattern node derivative u to pattern node derivative v is calculated as follows. The value of the link counter $\#_{(t(b(\mathsf{z})),t(b(\mathsf{u})),t(b(\mathsf{v})))}$ declared for the type $t(b(\mathsf{z}))$ of the pattern edge $b(\mathsf{z})$ restricted by direct types $t(b(\mathsf{u}))$ and $t(b(\mathsf{v}))$ of source and target pattern nodes $b(\mathsf{u})$ and $b(\mathsf{v})$, respectively, is divided by the value of the object counter $\#_{t(b(\mathsf{u}))}$ declared for the direct type $t(b(\mathsf{u}))$ of the source pattern node $b(\mathsf{u})$. Formally,

$$\forall \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SG}} : w_M(\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v}) = \frac{\#_{(t(b(\mathsf{z})),t(b(\mathsf{u})),t(b(\mathsf{v})))}(M)}{\#_{t(b(\mathsf{u}))}(M)}.$$

- The weight of NAC check edges is irrelevant.

As mentioned in the overview of search plan driven pattern matching in Sec. 4.2, in case of compiled graph transformation approaches, search plans are prepared for all binding combinations and pattern matching code fragments are generated and compiled for all search plans at compilation time. The current chapter also uses this schedule for search plan driven pattern matching. In this sense, as the following step in the search plan generation process, weighted search graphs get adorned for all necessary binding combinations at compile-time. For presentation purposes, our current investigation is restricted to the single adornment, in which all pattern node derivatives are free nodes.

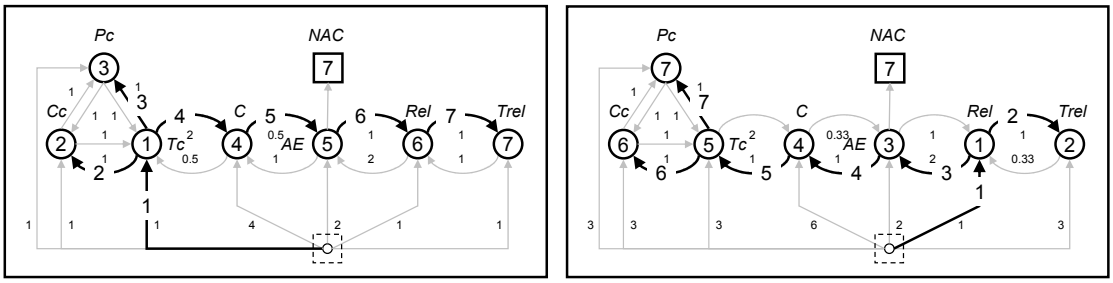
At this point, an adorned weighted search graph is available for each typical model selected by the domain engineer. A cost function is now defined for search plans to predict the performance of the pattern matching strategy driven by them.

The *cost of a search plan* (denoted by $c(\mathsf{SP})$) is an estimation for the number of nodes in the search space tree (SST), which would be generated during the execution of the pattern matching strategy 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. The number of nodes on the ith level of the SST is the product of branching factors of such search forest edges whose target node is labelled by at most i according to the search plan.

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 small. 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.

Example 24 Sample search plans on adorned, weighted search graphs are depicted in Fig. 7.4.

Both weighted search graphs are adorned by marking all their pattern node derivatives as free nodes, thus, only the dummy node is surrounded by the dashed box showing the bound part of the weighted search graph.



(a) Search plan defined for the adorned, weighted search graph of Fig. 7.3(c) whose statistics is based on Model1

(b) Search plan defined for the adorned, weighted search graph of Fig. 7.3(d) whose statistics is based on Model2

Figure 7.4: Sample search plans on adorned, weighted search graphs

Cost calculation is illustrated for the search plan of Fig. 7.4(a), which uses the statistics based on Model1 (Fig. 7.3(a)). This search plan binds table Tc first. As shown by the weight on the search forest edge leading to free node Tc with search plan label 1, a single object of type Table is expected to be found in Model1. As a consequence, the SST has one node on its first level. Since weights of search forest edges with labels 2 and 3 are also 1, the SST is expected to have $1 \cdot 1$ and $1 \cdot 1 \cdot 1$ node on its second and third level. Then the SST probably fork in two directions at level 4 as shown by the corresponding weight, thus, the number of nodes on this level is $1 \cdot 1 \cdot 1 \cdot 2$. By following the same procedure for the remaining search forest edges, and finally, by summing the SST nodes being found on different levels, the grand total number of nodes is resulted. On this specific search plan, the cost is $c(\mathsf{SP}) = 1+1\cdot 1+1\cdot 1\cdot 1+1\cdot 1\cdot 1\cdot 2+1\cdot 1\cdot 1\cdot 2\cdot 0.5+1\cdot 1\cdot 1\cdot 2\cdot 0.5\cdot 1+1\cdot 1\cdot 1\cdot 2\cdot 0.5\cdot 1\cdot 1 = 8$. The cost of the search plan of Fig. 7.4(b) whose statistics is based on Model2 of Fig. 7.3(b) is 12.

Note that the search plans of Fig. 7.4 are the ones that would be selected by the algorithms of Sec. 7.3.2, but their optimality even for their corresponding models cannot be generally proven. However, in specific cases like the search plan of Fig. 7.4(a) prepared for Model1 the optimality can be easily demonstrated.

- If the search plan started with node C, then the first and the second term in the sum would be 4 and 2, respectively. If the algorithm chose an edge with weight 1 at this point, the first three terms would already give 8. As a consequence, only the other edge with weight 0.5 could be selected at the third choice point. In this case, the sum of the first three terms is already 7, and there are four more edges with weight 1 to be included in the search plan, which exceeds 8.

- If the search plan started with node AE, then the edge from C to Tc with weight 0.5 should be included as soon as possible to decrease the term to be added to 1. This can happen in the third round at earliest. Even in this case, the cost is already 5, and there are four more edges with weight 1.

- If the search plan started with nodes Rel or Trel, then node Tc can only be reached via AE and C, which adds 5 to the cost at some point. Since the first edge has weight 1, and there are three additional similarly weighted edges, the cost of such search plans would be 9.

- The only remaining case is when the search plan starts with nodes on the left side (i.e., Tc, Pc, Cc), which yields a number of equivalent search plans whose cost is 8.

Definition 57 Given an adorned, weighted search graph ASG^M of the LHS pattern based on the statistics of model M together with a corresponding search plan SP, the **cost of search plan** SP (denoted

by $c(\mathsf{SP})$) is a non-negative number, which predicts the performance of the pattern matching strategy that is driven by search plan SP. The cost of search plan SP is calculated as

$$c(\mathsf{SP}) = \sum_{i=1}^{|V_{\mathsf{SG}}^F|} \prod_{j=1}^{i} w_j$$

where w_j denotes the weight $w_M(\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v})$ of search forest edge $\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SF}}$, which leads to free node v labelled by j according to search plan SP (i.e., $\mathsf{SP}(\mathsf{v}) = j$). The terms are summed for all free nodes V_{SG}^F of adorned search graph ASG^M.

7.3.2 Algorithms for finding low cost search plans

I adapted two traditional greedy algorithms to solve the problems of finding a low cost search forest for a given adorned, weighted search graph and a low cost search plan for a given search forest. Note that traditional minimum spanning tree algorithms operating on directed graphs 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 low cost search tree in a weighted search graph*, the Chu-Liu / Edmonds algorithm [26, 34] is used, which is outlined in Algorithm 7.1. 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.

Algorithm 7.1 The Chu-Liu / Edmonds algorithm

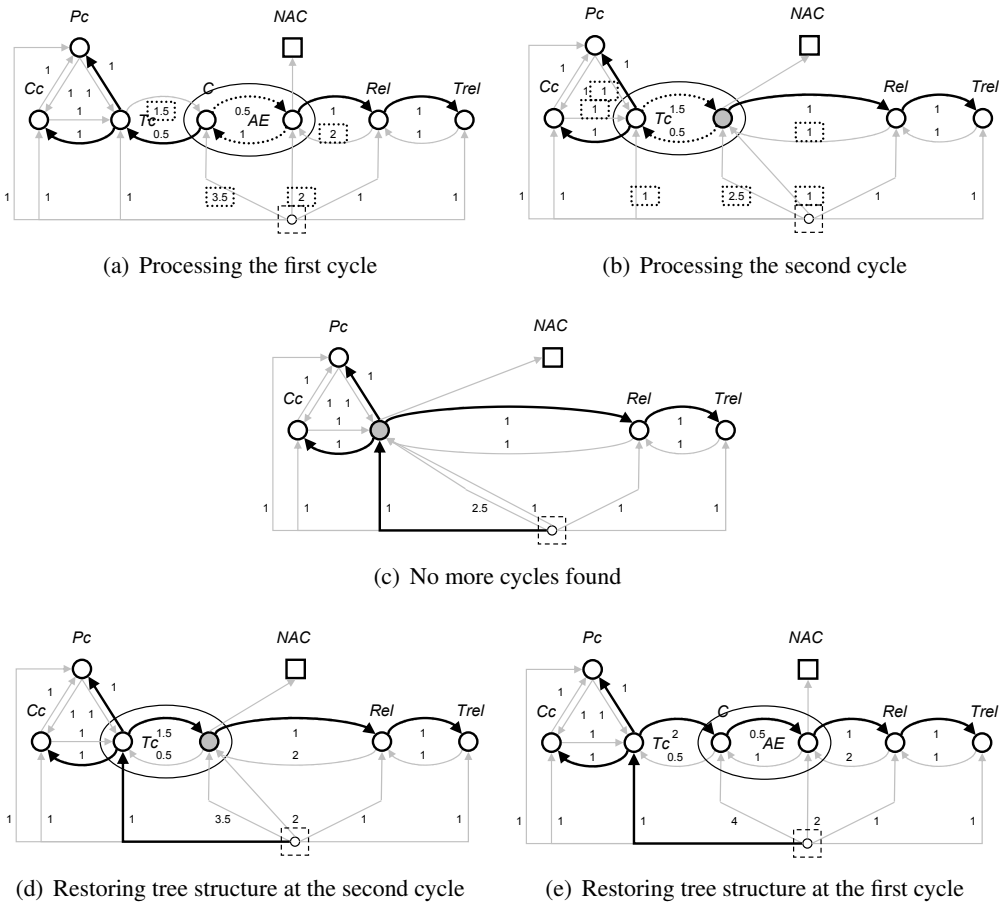
Given a weighted search graph.
Step 1: Discard the edges entering the dummy node or the bound nodes.
Step 2: For each free node, select an incoming edge with the smallest weight. Let the selected $n-1$ edges be the set S.
Step 3: If there are no cycles formed by the edges of S, then the selected edges constitute a minimum spanning tree of the graph and the algorithm terminates. Otherwise the algorithm continues.
Step 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.

$$w(i,k) = w(i,j) - [w(x(j),j) - min_l\{w(x(l),l)\}]$$

where $w(x(j),j)$ is the weight of the edge in the cycle which enters j.
Step 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.
Step 6: Go to step 3 with the contracted graph.

Example 25 Figure 7.5 presents how the Chu-Liu / Edmonds algorithm operates on the weighted search graph of Fig. 7.3(c).

Since the dummy node has only outgoing edges, and no bound nodes exist in the adorned weighted search graph, the first step is executed without discarding any edges. In the second step, for each free node, the incoming edge with the smallest weight is selected as shown by the black edges of Fig. 7.5(a). Since the circled part consisting of dotted edges bidirectionally connecting free nodes C and AE form



(a) Processing the first cycle

(b) Processing the second cycle

(c) No more cycles found

(d) Restoring tree structure at the second cycle

(e) Restoring tree structure at the first cycle

Figure 7.5: Cycle elimination in the Chu-Liu / Edmonds algorithm

a cycle, it has to be contracted. Weights of edges (denoted by numbers surrounded with dotted boxes) entering the contracted part have to be recalculated.

For the recalculation, the minimum weight in the cycle has to be determined first, which is 0.5 in our case. Then for each node of the cycle, the difference of the weight of the incoming cycle edge and the minimum value is calculated. For instance, the cycle edge leading to free node C has weight 1, thus, the difference is 0.5. Finally, weights of edges entering the contracted part should be decreased by the difference. By using this calculation method, the weight of edge connecting node Tc to C is reduced by 0.5 from 2 to 1.5, while the edge connecting nodes Rel to AE remains unmodified as the difference calculated for the edge entering AE is 0.

When weight recalculation is finished, the cycle gets contracted, and the edge with the smallest weight entering the contract node is selected into the forest. In our case, the navigation edge connecting free node Tc to C is chosen. When the first cycle is processed, the situation of Fig. 7.5(b) is resulted.

Since the new forest edge also forms a cycle together with the reverse navigation edge, the cycle eliminating procedure has to be repeated resulting in a situation shown in Fig. 7.5(c), which is now cycle free.

As the graph of Fig. 7.5(c) is already a forest, contracted nodes are now replaced with their origin cycles in reverse order. In this sense, the second cycle is restored first as shown by Fig. 7.5(d). While

replacing contracted nodes, the tree structure has to be restored by removing the cycle edge from the forest that leads to the same free node, to which the edge entering the cycle goes. In our case, the cycle edge leading to Tc is removed from the tree. Finally, the same procedure is repeated for the first cycle found, resulting in a search forest shown in Fig. 7.5(e).

In case of *finding a low cost search plan in a given search forest*, a simple greedy algorithm is used, which is sketched in Algorithm 7.2.

Algorithm 7.2 A greedy algorithm for generating a low cost search plan

---

Given a search forest.
Step 0: Set the counter to 1 and let S be the set consisting of the dummy node and the bound nodes.
Step 1: Select the smallest forest edge e that goes out from S.
Step 2: Set the target node of e to the value of the counter.
Step 3: Increment the counter by 1 and add the target node of e to S.
Step 4: If the search forest still has a node that is not in S, then go back to Step 1.

We do not state that these simple algorithms provide optimal solutions for our special cost model, but best engineering practice suggests that if edges with weights giving the minimum sum are selected, then the search forest 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.

Example 26 For illustrating the operation of Algorithm 7.2 on an example, the search forest of Fig. 7.5(e) has been selected as a starting point. In the first iteration, since the algorithm has no alternatives, it selects the iteration edge connecting the dummy node to free node Tc. In the second iteration, 3 outgoing forest edges can be found, namely the ones connecting node Tc to Cc, Pc, and C, respectively. These forest edges are processed in this specific order as their weights form a non-decreasing sequence,[1] and no new outgoing forest edges get into the set of valid choices. After the fourth iteration, the algorithm has no alternatives for forest edge selection, thus, it assigns labels 5, 6, and 7 as shown in Fig. 7.4(a), which in turn depicts the resulting search plan.

7.4 Compile-time tasks of adaptive pattern matching

At this point, several search plans have been elaborated by either the model-specific approach of Sec. 7.3, or any other search plan generation techniques.

In a traditional pattern matching approach, for each pair of LHS pattern and adornment, *only one* search plan is selected and compiled in the code generation phase resulting in executable code on the target platform. This approach is now extended to support adaptive execution by generating pattern matching code for several search plans having been prepared for each LHS pattern and adornment combination, and by defining a model-dependent cost function, which can be evaluated at run-time for estimating the performance of the given pattern matching strategy before actually executing it.

[1] Edges leading to nodes Cc and Pc can be processed in a reverse order as well as they have the same weight.

7.4.1 Theoretical foundations of compile-time support for adaptivity

Our solution for code generation uses the Strategy design pattern [48] (see Fig. 7.6), which means that a class extending the abstract `PatternMatchingStrategy` class is generated from each search plan.

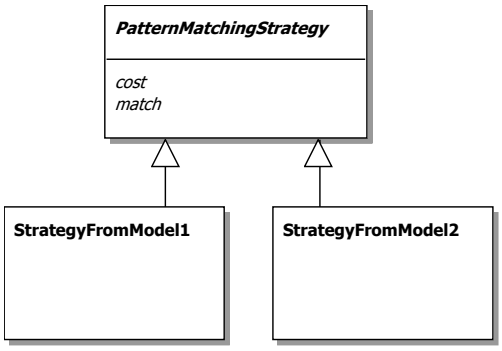


Figure 7.6: Strategy design pattern

The abstract class has two basic functionalities, which have to be supported by the generated classes as well.

(a) One method (i.e., `match()`) implements the actual pattern matching algorithm. Since our aim is to focus on the novelties of adaptation related topics, the generation of pattern matching algorithms is not discussed in more details that has already been presented in Sec. 4.2.5.

(b) The other relevant functionality is the calculation of cost for the given pattern matching strategy based on the statistics of the actual instance model available at run-time.

In order to support cost calculation functionality of pattern matching strategies, a model-dependent cost function $c_{\mathsf{SP}}(M)$ has to be specified for each search plan SP that drives the corresponding strategy. In this thesis, we use the cost function

$$c_{\mathsf{SP}}(M) = \sum_{i=1}^{\left|V^F_{\mathsf{SG}}\right|} \prod_{k=1}^{i} c_k(M)$$

where operation cost $c_k(M)$ is defined by the following three rules.

- **Cost of iteration.** If the source node d of the search forest edge leading to the free node v with label k is the dummy node, then the operation cost is given by the value of the object counter $\#_{t(v)}$ that has been defined for the direct type $t(v)$ of pattern node v, which is the origin of free node v, in turn. Formally,

$$\forall k \in \mathbb{Z}^+ \,:\, 1 \leq k \leq \left|V^F_{\mathsf{SG}}\right| \wedge \mathsf{SP}(\mathsf{v}) = k \wedge \mathsf{d} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SF}} \wedge b(\mathsf{v}) = v \Longrightarrow c_k(M) = \#_{t(v)}(M)$$

- **Cost of forward navigation.** If the search forest edge $\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v}$ connects pattern node derivative u to free node v with label k and goes in the *same* direction as its origin search graph edge $u \xrightarrow{z} v$,

then the operation cost is calculated as follows. The value of the link counter $\#_{(t(z),t(u),t(v))}$ declared for the type $t(z)$ of the pattern edge z restricted by direct types $t(u)$ and $t(v)$ of source and target pattern nodes u and v, respectively, is divided by the value of the object counter $\#_{t(u)}$ declared for the direct type $t(u)$ of the source pattern node u. Formally,

$$\forall k \in \mathbb{Z}^+ \,:\, 1 \leq k \leq \left|V^F_{\mathsf{SG}}\right| \wedge \mathsf{SP(v)} = k \wedge \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SF}} \wedge \mathsf{u} \in V^P_{\mathsf{SG}} \wedge b(\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v}) = u \xrightarrow{z} v \Longrightarrow c_k(M) = \frac{\#_{(t(z),t(u),t(v))}(M)}{\#_{t(u)}(M)}$$

- **Cost of backward navigation.** If the search forest edge $\mathsf{v} \xrightarrow{\mathsf{z}_{inv}} \mathsf{u}$ leading from pattern node derivative v to free node u with label k goes in the *opposite* direction as its origin search graph edge $u \xrightarrow{z} v$, then the operation cost is calculated as follows. The value of the link counter $\#_{(t(z),t(u),t(v))}$ declared for the type $t(z)$ of the pattern edge z restricted by direct types $t(u)$ and $t(v)$ of source and target pattern nodes u and v, respectively, is divided by the value of the object counter $\#_{t(v)}$ declared for the direct type $t(v)$ of the target pattern node v. Formally,

$$\forall k \in \mathbb{Z}^+ \,:\, 1 \leq k \leq \left|V^F_{\mathsf{SG}}\right| \wedge \mathsf{SP(u)} = k \wedge \mathsf{v} \xrightarrow{\mathsf{z}_{inv}} \mathsf{u} \in E_{\mathsf{SF}} \wedge \mathsf{v} \in V^P_{\mathsf{SG}} \wedge b(\mathsf{v} \xrightarrow{\mathsf{z}_{inv}} \mathsf{u}) = u \xrightarrow{z} v \Longrightarrow c_k(M) = \frac{\#_{(t(z),t(u),t(v))}(M)}{\#_{t(v)}(M)}$$

Note that the above-mentioned cost function is exactly the same as the one in Sec. 7.3.1 that has been used for defining model-specific search graphs for typical models in the optimization phase. However, the application schedule of these cost functions completely differ as the one defined in the current section is evaluated at run-time, while the other is calculated at compile-time. It is worth emphasizing that the presented adaptive pattern matching approach also requires the statistics support of the underlying model repository as it uses object and link counters for cost calculations.

7.4.2 Compile-time tasks in EJB3-based adaptive pattern matching

We present how an EJB3-based pattern matching engine can be made adaptive. In this sense, code fragments having been generated for a sample inherited concrete strategy class are shown, which implement pattern matching and cost calculation functionalities.

Example 27 Pattern matching driven by the search plan of Fig. 7.4(a) is implemented by the code presented in Listing 7.1. Note that this method is structurally similar to code fragments of Listings 4.1 and 4.2, which have been shown earlier in Sec. 4.2.5.

This strategy starts with iterating all tables and binding them to table Tc one-by-one (Lines 2–5). For each table in the model, a corresponding column and primary key is sought by navigating along links of type CF (Lines 6–9) and EO (Lines 10–13), respectively. At this point, the existence of a UF link between the previously bound column and primary key is checked (Lines 14–15). Then the class, which has been transformed to the table assigned to Tc is determined and it is bound to C (Lines 16–17). Note that reference edges can be navigated without a `while` loop as at most one multiplicity constraints have been defined for both ends of such edges in the metamodel. When AE, Rel, and Trel are already bound to a corresponding association end, association, and table by code fragments of Lines 18–21, 22–23, and 24–25, respectively, the checking of the NAC follows (Lines 26–29), which requires a matching to be initalized with the mapping of AE. When the NAC check fails, mappings of Tc, Cc, Pc, C, AE, Rel, and Trel constitute a complete matching, which can be returned as a result (Lines 31–35).

```
1  public Matching match(Matching initialMatching) {
2    // Level 1 -- Binds tc : Table
3    Iterator<Table> iTc = getAllTables();
4    while (iTc.hasNext()) {
5      Table tc = iTc1.next();
6      // Level 2 -- Binds cc : Column
7      Iterator<Feature> iCc = tc.getCF();
8      while (iCc.hasNext()) {
9        Column cc = (Column) iCc.next();
10       // Level 3 -- Binds pc : PKey
11       Iterator<ModelElement> iPc = tc.getEO();
12       while (iPc.hasNext()) {
13         PKey pc = (PKey) iPc.next();
14         // Checks UF edge
15         if (pc.getUF().contains(cc)) {
16           // Level 4 -- Binds c : Class
17           Class c = (Class) tc.getRef();
18           // Level 5 -- Binds ae : AssocEnd
19           Iterator<Feature> iAE = c.getSFT();
20           while (iAE.hasNext()) {
21             AssocEnd ae = (AssocEnd) iAE.next();
22             // Level 6 -- Binds rel : Association
23             Association rel = (Association) ae.getCF();
24             // Level 7 -- Binds trel : Table
25             Table trel = (Table) rel.getRef();
26             // Checks NAC
27             Matching mNAC = new Matching();
28             mNAC.set("AE", ae);
29             if (! nacMatcher.match(mNAC)) {
30               // Prepares the result matching
31               Matching result = new Matching();
32               result.set("Tc",tc); result.set("Cc",cc); result.set("Pc",pc);
33               result.set("C",c); result.set("AE",ae); result.set("Rel",rel);
34               result.set("TRel",trel);
35               return result;
36 } } } } } } }
```

Listing 7.1: Program code equivalent of the search plan of Fig. 7.4(a)

Example 28 By using the search plan of Fig. 7.4(a), a corresponding cost calculation method is also generated as shown by Listing 7.2.

The `cost` method processes search forest edges in an increasing order according to the label attached to their target free node. Local variable `term` stores the product of the cost of such forest edges that have already been processed, while variable `result` denotes the partial result of the addition. For each forest edge, variable `term` is multiplied by the corresponding cost of the edge, then it is added to the partial result. E.g., forest edge connecting Tc to Cc has label 2 at its target node derivative Cc, so it is processed in the second round (Lines 8–11). This forest edge represents a backward navigation along links of type CF starting from the table assigned to pattern node Tc, so its cost is calculated by dividing the link counter #(CF,Column,Table) by the object counter #(Table) declared for the direct type of the already fixed pattern node Tc.

```
1  public double cost() {
2    double term = 1.0;
3    double result = 0.0;
4    // Forest edge with label 1
5    // #(Table)
6    term *= cntTable();
7    result += term;
8    // Forest edge with label 2
9    // #(CF,Column,Table) / #(Table)
10   term *= cntCFColumnTable() / cntTable();
11   result += term;
12   // Forest edge with label 3
13   // #(EO,PKey,Table) / #(Table)
14   term *= cntEOPKeyTable() / cntTable();
15   result += term;
16   // Forest edge with label 4
17   // #(Ref,Class,Table) / #(Table)
18   term *= cntRefClassTable() / cntTable();
19   result += term;
20   // Forest edge with label 5
21   // #(SFT,Class,AssocEnd) / #(Class)
22   term *= cntSFTClassAssocEnd() / cntClass();
23   result += term;
24   // Forest edge with label 6
25   // #(CF,AssocEnd,Association) / #(AssocEnd)
26   term *= cntCFAssocEndAssociation() / cntAssocEnd();
27   result += term;
28   // Forest edge with label 7
29   // #(Ref,Association,Table) / #(Association)
30   term *= cntRefAssociationTable() / cntAssociation();
31   result += term;
32   return result;
33 }
```

Listing 7.2: Cost calculation program code based on the search plan of Fig. 7.4(a)

7.5 Run-time tasks of adaptive graph transformation

When compile-time tasks are completed, several pattern matching strategies are available in a compiled form for each LHS pattern and adornment combination. By means of model-dependent cost functions, these strategies also provide support for estimating their performance without actually executing them.

At run-time, when pattern matching is initiated for a given LHS pattern with a given adornment, the model-dependent cost of all corresponding strategies is calculated (see the Adapt phase in Fig. 7.1). Then the strategy with the best expected performance is executed (in the Exec phase of Fig. 7.1). Since the cost of a single strategy may vary depending on the current instance model, the relationship between costs of different strategies may change as transformation progresses. Note also that adaptivity only pays off, if the runtime gain caused by the execution of a low cost strategy exceeds the time spent on cost calculation, which is frequently the case for large models.

This observation also raises the problem of scheduling the invocations of run-time cost calculations. In this section, search plan costs are calculated before each rule application, however, this might not always be the best scheduling strategy as instance models typically do not change significantly during consecutive rule applications. An alternative (and sometimes better) strategy could be to only recalcu-

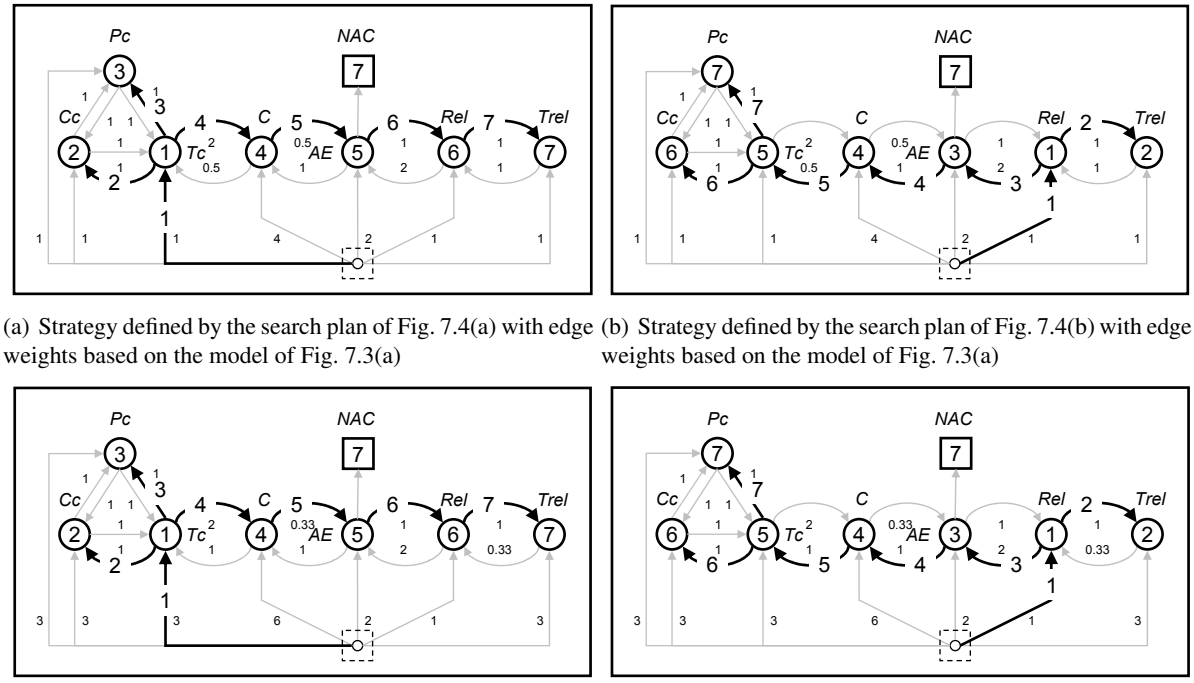
late search plan costs periodically, and to use the selected low cost search plan for rule applications of a certain, predefined length (e.g., 10 or 20). Note that the best scheduling strategy is typically application domain dependent, and this topic is not discussed further in this thesis.

7.5.1 Adaptive graph pattern matching: An illustrative example

Adaptivity in graph pattern matching is illustrated by an example.

Example 29 Let us consider that Model1 of Fig. 7.3(a) is evolved into Model2 of Fig. 7.3(b) as a result of some other rule applications. Let us further suppose that pattern matching strategies have been compiled for the search plans of Figures 7.4(a) and 7.4(b).

Cost calculation for possible combinations of pattern matching strategies and instance models is depicted in Fig. 7.7.



(a) Strategy defined by the search plan of Fig. 7.4(a) with edge weights based on the model of Fig. 7.3(a)

(b) Strategy defined by the search plan of Fig. 7.4(b) with edge weights based on the model of Fig. 7.3(a)

(c) Strategy defined by the search plan of Fig. 7.4(a) with edge weights based on the model of Fig. 7.3(b)

(d) Strategy defined by the search plan of Fig. 7.4(b) with edge weights based on the model of Fig. 7.3(b)

Figure 7.7: Illustrating adaptive graph pattern matching

In each subfigure, node labels and labelled black edges reflect the search plan which specified the pattern matching strategy, while edge weights depend on the corresponding instance model. Subfigures are organized in a tabular form based on the following guidelines.

- Rows represent points in the timeline of the transformation sequence being executed. Thus, each row shares a common instance model that is used by different strategies for cost calculation. As a consequence, edge weights are always the same in subfigures appearing in the same row.
- Each column represents a given strategy in different points of the transformation sequence. As a consequence, the structure of node labels and labelled black edges are always the same in subfigures appearing in the same column.

Initially, when Model1 is active, costs of strategies represented by Figures 7.7(a) and 7.7(b) are 8 and 9, respectively, thus, the first strategy is selected for execution. On the other hand, when the model has been evolved into Model2 by applying some other rules, a new situation appears, since costs of strategies represented by Figures 7.7(c) and 7.7(d) are now 21 and 12, respectively. As a result, the pattern matching engine executes the second strategy for Model2.

Note that this behaviour is not surprising as the first and second strategy have been prepared at compile-time in such a way that corresponding search plans are optimized for typical models Model1 and Model2, respectively.

7.5.2 Run-time tasks in EJB3-based adaptive pattern matching

We present how run-time tasks of an adaptive EJB3-based pattern matching engine (including tasks in the Adapt and Exec phases of Fig. 7.1) can be implemented.

For each LHS pattern and adornment combination, a stateless session bean is prepared, which maintains references to the available, compiled pattern matching strategies. The stateless session bean implements the `match()` method of the `IPatternMatcher` interface as presented by Listing 7.3.

```
1  boolean match(Matching m) {
2    // Finding the cheapest strategy
3    Strategy cheapestStrategy = strategies.get(0);
4    double cheapestCost = cheapestStrategy.cost();
5    for (int i=1; i < strategies.size(); i++) {
6      Strategy currentStrategy = strategies.get(i);
7      double currentCost = currentStrategy.cost();
8      if (currentCost < cheapestCost) {
9        cheapestCost = currentCost;
10       cheapestStrategy = currentStrategy;
11     }
12   }
13   // Executing the cheapest strategy
14   return cheapestStrategy.match(m);
15 }
```

Listing 7.3: Implementation of the `match()` method

The adaptation phase Adapt is reflected by Lines 2–12. This code fragment iterates over all pattern matching strategies (Lines 5–12), by also maintaining references to strategies (Line 6) and their corresponding costs (Line 7), and selects the cheapest strategy, which is later executed (in the Exec phase) in Lines 13–14.

By using container managed transactions, the `match()` method also provides the transaction context for pattern matching execution. This feature of the application server enables the parallel execution of a given rule without any modification in the pattern matching code.

7.6 Performance evaluation

In this section, the performance of the adaptive graph pattern matching technique and different EJB3-based transformation plugins is evaluated.

Performance analysis of the adaptive pattern matching approach. The quantitative performance analysis of the adaptive graph pattern matching approach has been deferred for years due to the lack of statistics support in underlying model repositories at the time of the development.

A recent report [12] examined the correlation of search plan costs and execution times in adaptive pattern matching approaches. More specifically, the authors focused on the quality of heuristics by checking whether low cost search plans also have short execution times.

In these experiments, three cost models have been examined and compared, namely, the heuristics proposed in this chapter, the highly similar original approach in GrGen [10, 11], and a new heuristics called Backtracking Lookup [12], which introduced costs for edge lookups as well. All search plans were generated for a given rule precondition according to each heuristics while measuring the execution time for each search plan. The exact experiment settings can be found in [12].

Based on the plots prepared for these measurements, the following statements could be made.

- On the Mutex STS test set of Sec. 5.3.1, all the heuristics (including the one presented in the current chapter) always produced the optimal search plan.

- The execution times of the possible search plans have a rather unbalanced distribution, so there is room for optimizations, which is confirmed by the development of the new Backtracking Lookup heuristics, which provided better search plans on an example originating from the compiler construction domain.

The memory consumption of graph transformation tools (including approaches using the adaptive pattern matching technique) has been assessed at the AGTIVE Tool Contest on the Sierpiński triangle benchmark example [133]. Note that this benchmark can only provide a inaccurate view on the performance of the adaptive technique as this approach should not have a significant influence on memory usage by its nature.

Quantitative performance analysis of transformation plugins. The performance of EJB3-based transformation plugins (namely, the portable EJB QL based solution of Sec. 6.6 and the approach of Sections 7.4.2 and 7.5.2 that operates on EJB3 entity beans) is evaluated by carrying out experiments on the object-relational mapping benchmark example, which has already been introduced in Sec. 5.4. Our main goal was to assess the overhead caused by an application server and the underlying persistence layer (including the relational database) required to run EJB3 applications. Therefore, after fixing a common pattern matching strategy, we executed measurements on three different approaches. In the first case, the transformation is performed on models consisting of in-memory Java objects without using an application server and a database. In the other two approaches, the transformation runs in an application server as an Enterprise Java Bean, and the corresponding instance model consists of entity beans, which are stored in the underlying database. The second approach executes pattern matching imperatively in the way described by the `match()` method in Sec. 7.4.2, while the third approach uses EJB QL queries in the pattern matching phase as presented in Sec. 6.6.

Since the optimization strategy of *parallel rule execution* is expected to have a significant impact on the run-time performance of all these approaches, its effect has also been observed during the measurements. Two test cases have been identified by switching this tool feature on and off. For each test case, the parameter N, which denotes the number of classes in the run, was fixed to 10, 30, 50, and 100.

For our measurements, we used a 1500 MHz Pentium machine with 768 MB RAM and a Linux operating system with kernel version 2.6.7 running Java SDK 1.5, JBoss application server version

4.2.1, Hibernate object-relational persistence layer version 3.2.4, and MySQL relational database version 4.1.7. The execution time results are shown in Table 7.1.

	ObjRel	Class	Model size	TS length	Java match	Java update	EJB match	EJB update	EJB QL match	EJB QL update
		#	#	#	msec	msec	msec	msec	msec	msec
ClassRule	parallel OFF	10	1342	146	0.01	0.03	117.31	13.91	7.14	125.87
		30	12422	1336	0.01	0.03	960.87	49.50	14.34	978.22
		50	34702	3726	0.05	0.06			28.07	2767.67
		100	139402	14951	0.01	0.01				
	parallel ON	10	1342	146	0.01	0.03	23.38	10.19	2.83	18.17
		30	12422	1336	0.01	0.01	41.07	6.33	2.66	38.97
		50	34702	3726	0.01	0.01	71.31	16.49	2.93	58.95
		100	139402	14951	0.01	0.01	123.45	13.04	3.40	115.48
AssociationRule	parallel OFF	10	1342	146	0.04	0.03	100.95	10.98	8.59	68.82
		30	12422	1336	0.03	0.01	796.00	25.84	7.73	478.06
		50	34702	3726	0.06	0.01			9.32	1364.83
		100	139402	14951	0.36	0.01				
	parallel ON	10	1342	146	0.01	0.03	7.94	5.39	2.30	5.41
		30	12422	1336	0.01	0.03	7.54	5.91	2.25	5.34
		50	34702	3726	0.01	0.02	10.79	5.29	2.30	7.71
		100	139402	14951	0.01	0.04	24.14	4.72	2.32	15.48
AssocEndRule	parallel OFF	10	1342	146	1.56	0.04	319.02	384.45	11.14	67.67
		30	12422	1336	0.39	0.01	2253.24	3387.53	12.91	148.55
		50	34702	3726	1.20	0.01			19.54	229.79
		100	139402	14951	5.11	0.01				
	parallel ON	10	1342	146	0.04	0.03	16.58	11.79	4.11	11.82
		30	12422	1336	0.02	0.01	51.10	43.32	4.08	34.47
		50	34702	3726	0.01	0.01	134.87	118.27	3.89	69.38
		100	139402	14951	0.01	0.01	402.06	358.67	3.95	215.93

Table 7.1: Performance evaluation of EJB3 plugins

The head of a row (i.e., the first two columns) shows the name of the rule and the optimization strategy settings for the single tool feature (i.e., parallel rule execution) on which the average is calculated. (Note that a rule is executed several times in a run.) The third column (Class) depicts the number of classes in the run, which is, in turn, the runtime parameter N of the test case. The fourth and fifth columns show the concrete values for the model size and the transformation sequence length, respectively. Heads of the remaining columns unambiguously identify the approach having been used. Values in match and update columns depict the average times needed for a single execution of a rule in the pattern matching and updating phase, respectively. Execution times were measured on a microsecond scale, but a millisecond scale is used in Table 7.1 for presentation purposes. Light grey areas denote run-time failures due to exceeding the default limits of the application server on the number and size of transactions.

Our observations can be summarized as follows.

- **General performance related observations.** On a given model size, the pure Java solution runs three or four orders of magnitude faster than the enterprise applications during both the pattern matching and the updating phase. The slowness of enterprise applications is caused mainly by the overhead of disk-based storage, and transaction handling, and partially by the maintenance of database connections.

- **Performance degradation caused by the different data structures used by the model repository.** There is a further factor, which causes a significant deceleration of enterprise applications during the updating phase and in case of large models. This originates from the fact that the most frequently applied model manipulation steps operate on different data structures. In this sense, the pure Java solution stores neighbouring objects in linked lists, to which new objects can be added in constant time, while the underlying relational database of application servers uses a tree-based data structure, in which additions can only be performed in logarithmic time.

- **Unsuitable proxy and transaction handling mechanisms in Hibernate and JBoss.** An unacceptably slow execution time and high resource demand have been detected in case of enterprise applications, which have been caused by the current version of the proxy mechanism of Hibernate and the transaction handling of JBoss.

7.7 Conclusions

In the current chapter, I elaborated an adaptive method for executing model-specific search plans in order to improve the performance of graph transformation in its pattern matching phase.

- *Costs and optimization mechanisms for model-specific search plans.* I defined a cost function for model-specific search plans which estimates the size of the search space that would be traversed during search plan execution. In order to find a low cost search plan according to the special cost function defined for model-specific search plans, I elaborated an optimization technique by customizing traditional greedy algorithms (Sec. 7.3).
- *Adaptive graph transformation engine.* I elaborated an adaptive graph transformation engine which is able to select the optimal pattern matching strategy at execution time from the set of precompiled strategies by exploiting run-time model statistics (Sections 7.4.1 and 7.5.1).
- *EJB3-based prototype engine.* I prepared an EJB3-based prototype of the adaptive graph transformation engine for the Java 2 Enterprise Edition (J2EE) platform by generating code for pattern matching and cost calculation functionalities of concrete strategies, and by implementing a stateless session bean that selects the optimal strategy at run-time (Sections 7.4.2 and 7.5.2).
- *Quantitative evaluation of Java, EJB3 and EJB QL based pattern matching.* After fixing a common search plan, I examined and compared the efficiency of Java, EJB3 and EJB QL based pattern matching implementations on a benchmark example (Sec. 7.6).

The above-mentioned results are published in [7, 63, 64, 148, 152, 156]. The results of this chapter have been integrated into the VIATRA2 model transformation framework [142, 143, 144].

Relevance

It is worth noting that the main contributions of the current chapter (i.e., model-specific search plans and the adaptive pattern matching technique) constitute an orthogonal framework of novelties so their relevance can be evaluated independently and also in combination.

Concepts of model-sensitive search plans are directly applicable to further fine-tune the performance of any compiler-based GT approaches. This statement is confirmed in [51, 132] by the developers of GrGen tool who independently develop the same technique [10, 11] with minor differences in operation cost assignment and search plan cost calculation. A recent paper [12] reported about a new heuristics called Backtracking Lookup by introducing costs for edge lookups, which may further

accelerate pattern matching in scenarios originating from compiler construction application domain. GrGen generates search plan driven strategies as C# code, in contrast to our approach, which produces EJB3 compliant Java code. GrGen has no predefined scheduling strategies for cost recalculation. This task can be initiated on request from the GrShell. Each time a search plan cost is recalculated, a corresponding pattern matching strategy is generated, which can later be dynamically linked into GrGen. (For comparisons to sophisticated pattern matching strategies of compiled (but non-model-sensitive) graph transformation approaches see Sec. 7.1.)

The combination of adaptive and model-sensitive pattern matching techniques can be used in interpreted GT engines as well. If model-specific search graphs are used, then a single low cost search plan can be dynamically prepared at run-time at each rule invocation by using the current adornment and the statistics of the instance model under transformation. In this sense, the pattern matching is always guided by a low cost strategy with respect to the current instance model in contrast to compiled GT engines, which can only select the optimal from the set of precompiled pattern matching strategies. As a common practical application field of this combined technique, the model transformation framework of VIATRA2 should be mentioned as all these contributions are currently being built into its new interpreted graph transformation engine.

The adaptive, model-sensitive pattern matching technique has been recently analyzed quantitatively in [12] by using the benchmarking framework of Chapter 5. In this survey, the authors prove the feasibility of the adaptive approach by demonstrating a strong correlation between search plan costs and pattern matching execution time, which means that the cost model having been presented in Sec. 7.4.1 can be successfully used for estimating the performance of pattern matching.

In the future, adaptivity and model-specific search plans are aimed to be integrated into the generic framework [64] of search plan operations to make its cost assignment dynamic and model dependent by replacing the current static, heuristics-based method. In this sense, this generic framework enables the proper and performance optimal positioning of complex search plan operations (such as NAC checking and the invocation of recursive patterns [152]).

Limitations

It should be emphasized that this chapter presented *practical heuristics* for pattern matching. Neither the technique of model-specific search plans, nor the approach of adaptive graph pattern matching can be provenly optimal by their nature due to the following reasons.

First of all, counters of the model repository store aggregated statistical data, which obviously cannot reflect the exact structure of the instance model. Even if all counters share the very same values, the underlying instance models can produce significantly different number of matchings. As a direct consequence, a low cost search plan cannot guarantee a small search space tree during execution. As counters are used both at compile-time and at run-time for search plan optimization and for cost calculation, respectively, the pattern matching strategy that is executed finally in the Exec phase is not necessarily optimal. Refining data structures that store statistics in the model repository can be an obvious way to improve the precision of both techniques, but such an approach can easily become unfeasible due to increased storage or computation efforts caused by the refinement. Note that the success of adaptive graph pattern matching is highly sensitive to the time spent on cost calculation, which is based on the data structures of the repository.

The other problem stems from the special cost function that has been defined for search plans. The customized greedy algorithms can only provide low cost, but not necessarily optimal search plans. From a pure mathematical point of view, it is easy to find counterexamples for the optimality of the

presented algorithms, but these examples are rarely produced by real-life application domains and even in such cases the generated low cost search plans provide good solutions.

Finally, as a general guideline, it can be stated that adaptivity only pays off, if the runtime gain caused by the execution of a low cost strategy exceeds the time spent on cost calculation.

CHAPTER

8 Incremental Graph Transformation

In this chapter, I present the foundations of an incremental graph pattern matching engine for handling rules with negative application conditions, which keeps track of existing matchings in an incremental way to reduce the execution time of graph pattern matching.

8.1 Motivation

Despite the large variety of existing graph transformation tools, the implementation of their graph transformation engine typically follows the same principle. First a matching occurrence of the left-hand side of the graph transformation rule is searched by some sophisticated graph pattern matching algorithm. Then potential negative application conditions are checked that might eliminate the previous occurrence. Finally, the engine performs some local modifications to add or remove graph elements to the matching pattern, and the entire process starts all over again.

As the information on a previous matching is lost when a new transformation step is initiated, the complex and expensive graph pattern matching phase is restarted from scratch each time. This non-incremental behaviour can be a performance bottleneck as demonstrated e.g., by our benchmarking experiments [154] and by practical experience in model-based tool integration [80] based on triple graph grammars [120].

Related work

Incremental updating techniques have been widely used in different fields of computer science. Now a brief overview is given on incremental techniques that could be used for graph transformation.

- **Rete networks.** [22] proposed an incremental graph pattern matching technique based on the idea of Rete networks [45], which stems from rule-based expert systems. In their approach, a network of nodes is built at compile time from the LHS graph to support incremental operation. Each node performs simple tests on the entities (i.e., nodes, edges, partial matchings) arriving to its input(s). If the test succeeds, the node groups entities into compound ones, which are then sent downwards in the network. On the top level of the network, there are nodes with a single input that let such objects and links of a given type to pass that have just been inserted to or

removed from the model. On intermediate levels, network nodes with two inputs appear, each representing a subgraph of the LHS. These nodes try to build matchings for the subgraph from the smaller matchings located at the inputs of the node. On the lowest level, the network has terminal nodes, which do not have outputs. They represent the entire LHS graph. Entities reaching the terminals represent complete matchings for the LHS.

- **PROGRES.** The PROGRES [122] graph transformation tool supports an incremental technique called attribute updates [66]. At compile-time, an evaluation order of LHS nodes is fixed by a dependency graph. At run-time, a bit vector is maintained for each object expressing whether it can be bound to the nodes of the LHS. When objects are deleted, some validity bits are set to false according to the dependency graph denoting the termination of possible partial matchings. In this sense, PROGRES performs immediate invalidation of partial matchings. On the other hand, validation of partial matchings are computed on request (i.e., when a matching for the LHS is requested).

- **TefKat.** TefKat [82] is a declarative model transformation language together with an execution engine implemented as an Eclipse plugin. The transformation engine performs an SLD resolution based interpretation during which a search space tree is constructed to represent the trace of transformation execution. This tree is maintained incrementally in consecutive steps of transformations as described in [59].

- **View updates.** In relational databases, materialized views, which explicitly store their content on the disk, can be updated by incremental techniques. Counting and DRed algorithms [57] first calculate the delta (i.e., the modifications) for the view by using the initial contents of the view and base tables and the deltas of base tables. Then the calculated deltas are performed on the view.

These techniques only provide partial solutions for typical model transformation problems as PROGRES supports pattern matching in such cases when the rule precondition is a connected graph [98], while the Rete-based approach lacks the support for negative application conditions and inheritance.

Objectives

I propose foundational data structures and algorithms for incremental graph pattern matching where all complete matchings (and also non-extensible partial matchings) of a rule are stored explicitly in a tree according to a given search plan. This tree is updated incrementally triggered by the modifications of the instance model. Negative application conditions are handled uniformly by storing all matchings of the corresponding patterns. Additionally, we keep track if a matching of the negative condition pattern invalidates the matching of the positive pattern. Furthermore, as the main conceptual novelty, we introduce a notification mechanism by maintaining registries for quickly identifying those partial matchings, which are candidates for extension or removal when an object or a link is inserted to or deleted from the model.

Architectural overview

In Figure 8.1, an architectural overview is provided on the envisaged workflow of an incremental pattern matching engine. Note that a main driver of this architecture is to allow easy adaptation to existing GT engines.

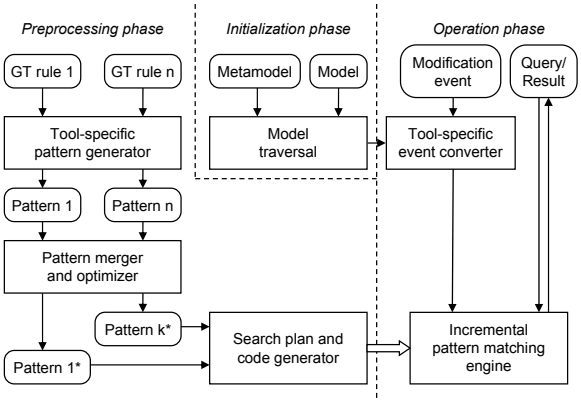


Figure 8.1: Architectural overview of incremental pattern matching

Preprocessing. In a preprocessing phase, patterns are first extracted from graph transformation rules (based upon the LHS and NAC of the rules). Since these patterns may be overlapping, this initial set of graph patterns can be optimized by merging them along common parts to be maximized and by extending overlapping areas to original individual patterns. Afterwards, search plans are derived for the optimized pattern set, and template-based code generation is applied to implement the matching tree tailored to the actual GT rules.

Initialization. In the initialization phase, the tree is constructed based upon a given initial model and its metamodel. While this initialization step can be time consuming, this is only performed once, prior to the actual transformations.

Operation. In the operation phase (which is the main focus of the current chapter), the incremental pattern matching engine listens to the notifications sent by the GT engine on model modifications, and keeps track of the changes in the tree. As a consequence, pattern matching queries coming from the GT engine are executed in constant time.

The basic structure of the current chapter is the following.

- Section 8.2 introduces few new concepts to ease the presentation of incremental pattern matching.
- Section 8.3 presents data structures needed for maintaining, efficiently storing, invalidating, and notifying partial matchings, and for accelerating the retrieval of complete matchings.
- Section 8.4 first presents the core algorithm of the incremental pattern matcher, which is invoked whenever the model is changed, then it demonstrates the incremental operation on an example, and finally, details of the main modification event handler methods are reviewed.
- Section 8.5 assesses the computational efficiency of the incremental approach, and compares it to the run-time performance of FUJABA using the object-relational mapping of Section 5.4 as a benchmark example.
- Section 8.6 presents an alternative method for incremental graph transformation by persistently storing partial matchings in tables of an underlying relational database and by executing SQL commands for reconstructing the database content based on the modifications of the instance model.
- Section 8.7 concludes this chapter with summarizing its relevance.

8.2 Concepts for supporting incremental pattern matching

By using the terms defined in Sections 2.2, 3.1, and 4.2, few new concepts are now introduced to ease the presentation of incremental pattern matching.

The *kth partial plan of a search plan* is the subgraph of the corresponding adorned search graph induced by the dummy node, all bound nodes, and the free nodes labelled by the k smallest positive integers. Search plan labels are preserved for the partial plans as well.

Recall that there is a one-to-one correspondance between nodes in a pattern and their derivatives in the corresponding (adorned) search graph. The term *kth subpattern* is used for the subgraph of the pattern being induced by the pattern node counterpart of the derivatives of the kth partial plan. The term *kth pattern node* is used for the pattern node, whose derivative is labelled by k according to corresponding search plan. Incoming and outgoing edges of the kth pattern node are referred informally as *incoming and outgoing condition edges*, respectively.

Example 30 A search plan (Fig. 8.2(b)) is defined for the LHS pattern of ClassRule (Fig. 8.2(a)) by fixing the (1) C, (2) P, (3) S order on pattern node derivatives. A search plan for the NAC pattern (not shown in Figure 8.2) is (1) T, (2) C.[1] The first, second, and third partial plan of the search plan of Fig. 8.2(b) are shown by the unshaded parts of Figures 8.2(d), 8.2(f), and 8.2(h), while corresponding subpatterns LHS_1, LHS_2, and LHS_3 are depicted by Figures 8.2(c), 8.2(e), and 8.2(g), respectively.

The first, the second, and the third pattern node are represented as nodes with thick lines in Figures 8.2(c), 8.2(e), and 8.2(g), respectively, while their corresponding derivatives are denoted by dotted free nodes in the figures on the right. The Ref edge (denoted by a thick line) connecting pattern node P to S represents an incoming condition edge in Fig. 8.2(g).

The formalization of the concepts is now presented.

Definition 58 Given a search plan SP defined for an adorned search graph ASG, the **kth partial plan SP_k of a search plan** SP is the subgraph of adorned search graph ASG induced by such search graph nodes, which are excluded from the set of NAC nodes, and which are labelled by integers not exceeding k according to search plan SP. Formally, $\mathsf{SP}_k \subseteq \mathsf{ASG}$ such that $\forall \mathsf{x} \in V_{\mathsf{SG}} : \big(\mathsf{x} \in V_{\mathsf{SP}_k} \Longleftrightarrow \mathsf{x} \in V_{\mathsf{SG}} \setminus V_{\mathsf{SG}}^{\mathsf{NAC}} \wedge \mathsf{SP}(\mathsf{x}) \leq k\big)$, and $\forall \mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SG}} : \Big(\mathsf{u} \xrightarrow{\mathsf{z}} \mathsf{v} \in E_{\mathsf{SP}_k} \Longleftrightarrow \mathsf{u} \in V_{\mathsf{SP}_k} \wedge \mathsf{v} \in V_{\mathsf{SP}_k}\Big)$.

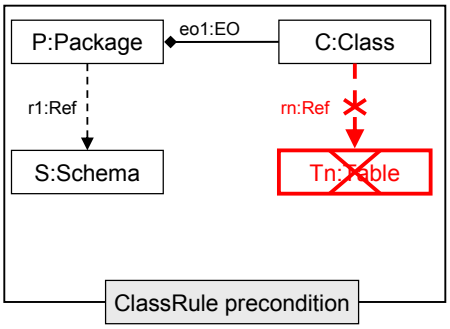
In other words, the node set of the kth partial plan consists of the dummy node, all bound nodes, and such free nodes that have the k smallest positive integers as labels.

Definition 59 Given a search plan SP defined for the adorned search graph ASG of pattern graph G, the **kth subpattern G_k of pattern graph** G is the subgraph of G induced by the origin pattern nodes of such derivatives, which also appear in the kth partial plan SP_k of search plan SP. Formally, $G_k \subseteq G$ such that $\forall x \in V_G : \big(x \in V_{G_k} \Longleftrightarrow \big(\exists \mathsf{x} \in V_{\mathsf{SG}}^P \cap V_{\mathsf{SP}_k} : b(\mathsf{x}) = x\big)\big)$, and $\forall u \xrightarrow{z} v \in E_G : \Big(u \xrightarrow{z} v \in E_{G_k} \Longleftrightarrow u \in V_{G_k} \wedge v \in V_{G_k}\Big)$.

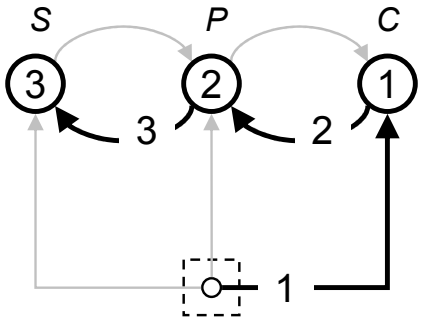
Consequently, if a pattern graph G has n nodes to be matched during pattern matching (i.e., its adorned search graph has n free nodes), then pattern graph G has n+1 subpatterns, namely, $G_0, \ldots, G_n$.

Definition 60 Given a search plan SP defined for the adorned search graph ASG of pattern graph G, the **kth pattern node v_k of pattern graph G according to the search plan** SP is the pattern node, whose derivative v_k has label k according to search plan SP. Formally, $b(\mathsf{v}_k) = v_k \wedge \mathsf{SP}(\mathsf{v}_k) = k$.

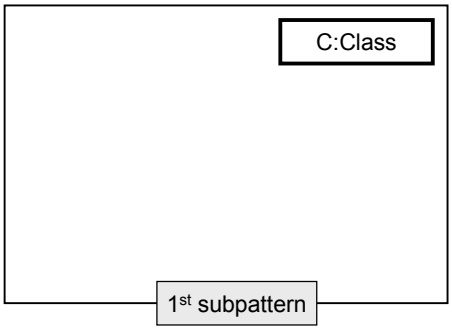
[1] Search plans of the current example have been selected manually for presentation purposes.



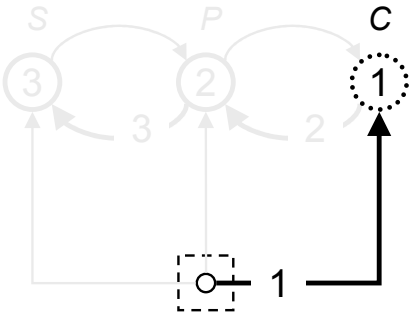
(a) Precondition of ClassRule



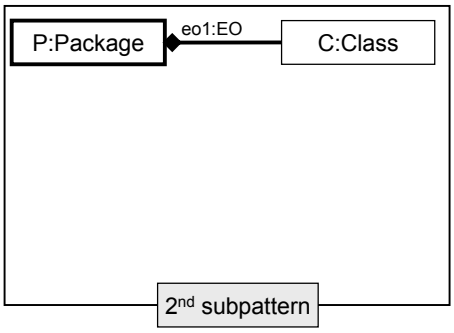
(b) A search plan prepared for the LHS of ClassRule



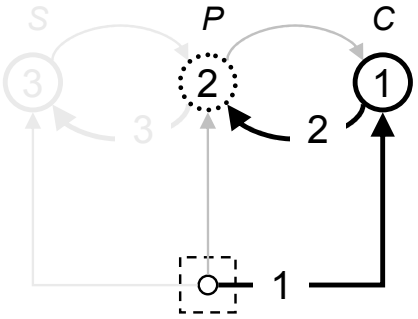
(c) The first subpattern LHS_1 of the LHS



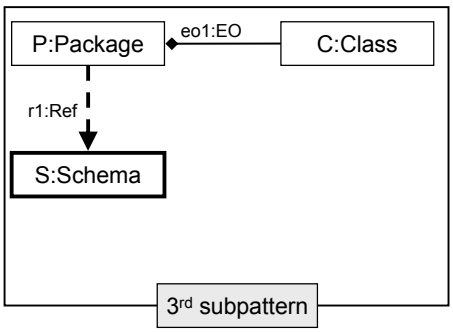
(d) The first partial plan of Fig. 8.2(b)



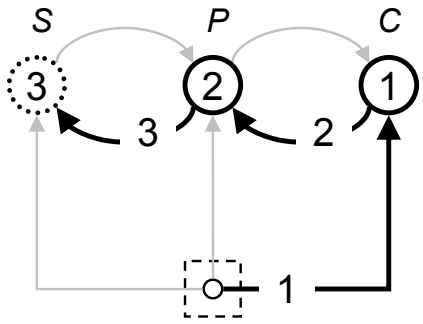
(e) The second subpattern LHS_2 of the LHS



(f) The second partial plan of Fig. 8.2(b)



(g) The third subpattern LHS_3 of the LHS



(h) The third partial plan of Fig. 8.2(b)

Figure 8.2: Subpatterns and partial plans

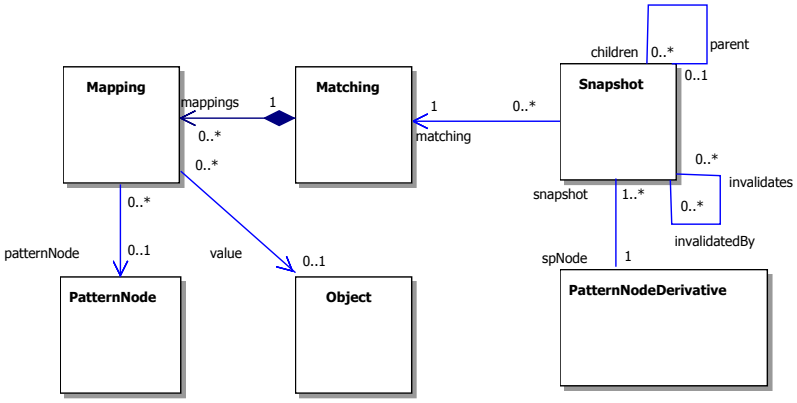
8.3 Data structures for incremental pattern matching

In this section, I present data structures needed for (i) maintaining, (ii) efficiently storing, (iii) invalidating, and (iv) notifying partial matchings, and for (v) accelerating the retrieval of complete matchings. Algorithms of the incremental pattern matching engine, which operate on these data structures are discussed later in Sec. 8.4.

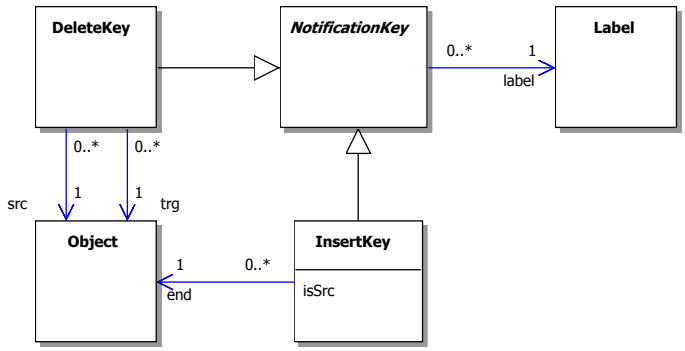
Class diagrams depicting the different aspects of data structures being used by the incremental pattern matching engine are shown in Fig. 8.3.

8.3.1 Matching snapshots and snapshot trees

The concept of snapshots has already been introduced for denoting matchings for subgraphs (i.e., partial matchings for patterns) in Sec. 4.1.1, when pattern matching algorithms have been discussed in general. Since the incremental algorithm is search plan driven, the interpretation of *snapshots* (denoted by numbered circles in Fig. 8.4) is now restricted by only allowing matchings for subpatterns, which constitute a special subset of subgraphs based on the currently active search plan. However, in any other respects, this new definition still fully complies to the original concept.



(a) Matchings



(b) Event processing

Figure 8.3: Data structures of the incremental pattern matching engine

Since a snapshot is a logical representation of a (partial) matching, there is a one-to-one correspondance between these terms, and consequently, these names could have been used interchangeably. However, in the following, the term snapshot is used specifically for the data structure appearing in the incremental algorithm, while the word (partial) matching denotes the actual (partial) morphism between the pattern graph and the model.

To support incremental behaviour, a *snapshot tree* is maintained for each pattern graph, which consists of snapshots being organized into a tree structure along *parent-child edges* (depicted by dashed arcs in Fig. 8.4). The root of the tree (i.e., the single node on the first level) denotes the initial matching, in which pattern nodes are only bound outside the pattern matching algorithm, and not by the algorithm itself. Snapshots denoting matchings for a given subpattern can always be found on the same *level* of the tree (marked by light grey areas in Fig. 8.4). The mapping of subpatterns to tree levels is guided by the search plan having been fixed for the pattern graph. A tree node on the (k+1)th level (i.e., having distance k from the root) represents a matching for the kth subpattern being specified by the search plan. Each *leaf* represents a maximal partial matching for the pattern. If the pattern has n nodes to be matched, then each leaf on the (n+1)th (i.e., deepest possible) level represents a (complete) matching.

Example 31 Sample models of Figures 8.4(c), 8.4(e), and 8.4(g) and the corresponding data structure contents are presented in Figures 8.4(d), 8.4(f), and 8.4(h), respectively. Figures 8.4(d), 8.4(f), and 8.4(h) show snapshot trees in their top-right corner, they depict binding arrays at the bottom, while notification arrays are presented in their left part.

Fig. 8.4(f) contains two snapshot trees representing the partial matchings of the LHS and the NAC pattern, respectively. Snapshots 1 and 2 denote empty matchings. Snapshot 3 is located on the second level of the tree defined for the LHS pattern, thus, it is a matching for the first subpattern LHS_1, which contains a single mapping that assigns object c1 to pattern node C. Snapshot 3 is a child of snapshot 1, as the matching represented by the latter can be extended by the mapping of pattern node C.

In the context of Fig. 8.4(d), snapshot 3 is a maximal partial matching as it cannot be further extended, due to the lack of outgoing EO edges leading out of class c1. On the other hand, snapshot 3 is not a maximal partial matching in Fig. 8.4(f) as it can be extended e.g., by mappings P to p and S to s to get the matching represented by snapshot 5. This means a (complete) matching for the LHS pattern as snapshot 5 is located on the lowest tree level LHS_3.

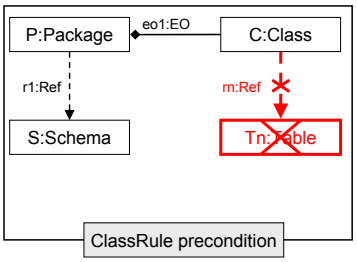
Definition 61 The **snapshot universe** $\mathbb{S}_G$ denotes all possible partial matchings for a pattern graph G.

Definition 62 Given a search plan SP defined for the adorned search graph ASG of a pattern graph G and a model M, a **snapshot** s_{G_k} **of pattern graph** G is a logical representation of a matching m_{G_k} for the kth subpattern G_k of pattern graph G in model M. Formally, $s_{G_k} \in \mathbb{S}_G$

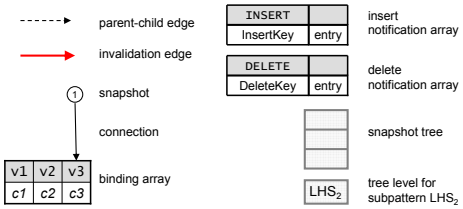
In the following, superscript m is used for identifying the matching represented by a snapshot.

Definition 63 Given a search plan SP defined for the adorned search graph ASG of a pattern graph G and a model M, a **snapshot tree** $\mathbb{ST}_G = \left(\mathbb{S}_G^{\mathcal{T}}, r_G, p_G\right)$ is a data structure described by a triple consisting of the following parts.

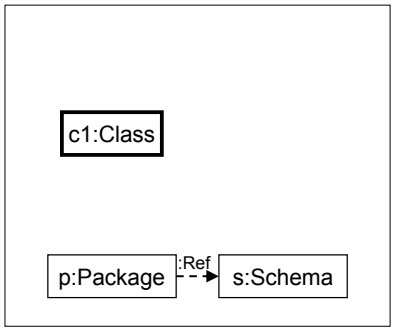
- **Tree nodes** $\mathbb{S}_G^{\mathcal{T}}$ are snapshots of the pattern graph G. Formally, $\mathbb{S}_G^{\mathcal{T}} \subseteq \mathbb{S}_G$.
- The **root node** r_G is a tree node, which represents the matching m_{G_0} for the 0th subpattern G_0 of pattern graph G in model M. Formally, $\exists r_G \in \mathbb{S}_G^{\mathcal{T}} : r_G^m = m_{G_0}$.
- The **parent function** $p : \mathbb{S}_G^{\mathcal{T}} \rightarrow \mathbb{S}_G^{\mathcal{T}}$ defines the parent node of each snapshot.



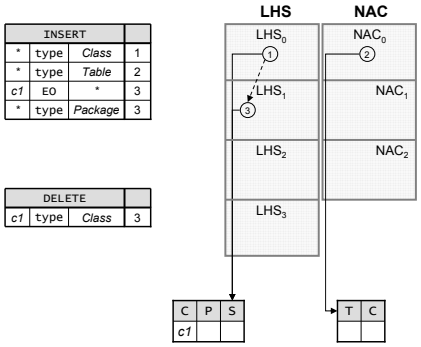
(a) Precondition of ClassRule



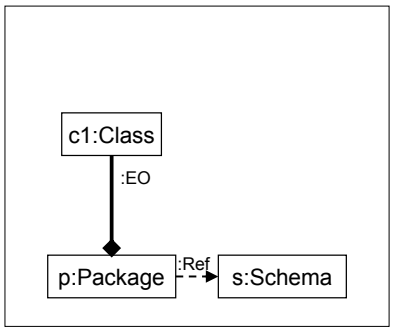
(b) Notational guide for data structures



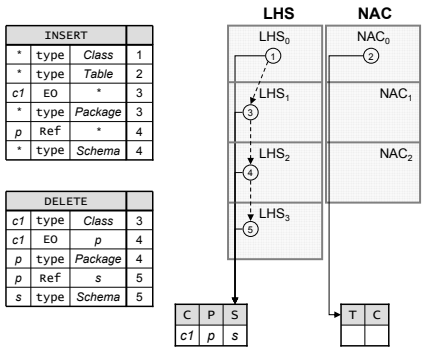
(c) Model 1



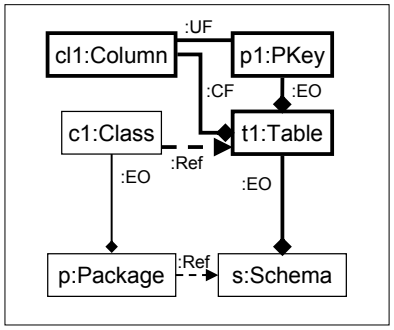
(d) Data structure contents for Model 1



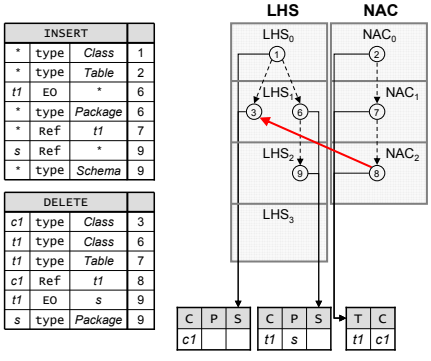
(e) Model 2



(f) Data structure contents for Model 2



(g) Model 3



(h) Data structure contents for Model 3

Figure 8.4: Sample models and the corresponding data structures

The following restrictions ensure the well-formedness of the snapshot tree.

- Each tree node s except for the root r_G has a parent. Formally, $\forall s \in \mathcal{S}_G^{\mathcal{T}} \setminus \{ r_G \}, \exists t \in \mathcal{S}_G^{\mathcal{T}} : p(s) = t$, and $\nexists s \in \mathcal{S}_G^{\mathcal{T}} : p(r_G) = s$.
- If tree node s_{G_k} is not the root r_G, and it represents a matching m_{G_k} for the kth subpattern G_k of pattern graph G in model M, then its parent $s_{G_{k-1}}$ must represent a matching $m_{G_{k-1}}$ for the (k–1)th subpattern G_{k-1} of pattern graph G in model M. Formally,

$$\forall k \in \mathbb{N} : 1 \leq k \leq \left|V_{\mathsf{SG}}^F\right| \Longrightarrow \Big(\forall s_{G_k} \in \mathcal{S}_G^{\mathcal{T}} \setminus \{ r_G \}, \forall m_{G_k} : s_{G_k}^m = m_{G_k} \Longrightarrow \Big(\exists s_{G_{k-1}} \in \mathcal{S}_G^{\mathcal{T}}, \exists m_{G_{k-1}} : s_{G_{k-1}}^m = m_{G_{k-1}} \wedge p(s_{G_k}) = s_{G_{k-1}}\Big)\Big).$$

8.3.2 Binding arrays

In implementations, matchings are physically stored as one-dimensional *binding arrays*, which are indexed by the pattern nodes. An entry in a binding array stores pattern node–object pairs in the corresponding matching. When one matching is an ancestor of another one, their binding arrays can be shared in order to reduce memory consumption as the ancestor matching contains a subset of the mappings of the descendant matching. Consequently, for each pattern graph G with n variables, a binding array `match[n]` of size n is used. In Fig. 8.4, binding arrays are connected to snapshots by solid black lines.

Example 32 Since the LHS pattern has 3 nodes, snapshots of the LHS snapshot tree refer to binding arrays having 3 entries as it is shown e.g., in the lower part of Fig. 8.4(f). Each column of the binding array of the LHS snapshot tree represents a mapping, which shows the object (in the lower row) to which the pattern node (in the upper row) has been mapped. Note that the array that contains mappings C to c1, P to p, and S to s can be shared by snapshots 1, 3, 4, and 5, as they only consist of mappings of the first 0, 1, 2, and 3 free nodes, respectively.

8.3.3 Invalidation edges

Invalidation edges (denoted by thick (red) arcs) represent the invalidation of partial matchings of a LHS caused by (complete) matchings of a NAC.

Example 33 The red invalidation edge of Fig. 8.4(h) connecting snapshots 7 to 3 means that snapshot 7 represents a (complete) matching for the NAC pattern, which invalidates the partial matching of snapshot 3 as both map the shared node C to the same object c1. As long as snapshot 3 is invalidated (as shown by the incoming invalidation edge), it cannot be part of a (complete) matching for the LHS pattern, which fact is marked by the empty subtree rooted at snapshot 3.

Definition 64 Given a model M and snapshot universes $\mathcal{S}_{\mathsf{LHS}}$ and $\mathcal{S}_{\mathsf{NAC}}$ defined for pattern graphs LHS and NAC, respectively, by also using search plans defined for the corresponding adorned search graphs, a **snapshot** s_{NAC} **invalidates snapshot** s_{LHS_k} (denoted by $s_{\mathsf{NAC}} \rightarrowtail s_{\mathsf{LHS}_k}$), if all shared nodes of NAC and LHS are mapped by matchings s_{NAC}^m and $s_{\mathsf{LHS}_k}^m$ defined for the NAC and the kth subpattern LHS_k of pattern graph LHS, respectively, and each shared node is mapped to the same object in model M by both matchings. Formally, $\forall s_{\mathsf{LHS}_k} \in \mathcal{S}_{\mathsf{LHS}}, \forall s_{\mathsf{NAC}} \in \mathcal{S}_{\mathsf{NAC}} : \Big(s_{\mathsf{NAC}} \rightarrowtail s_{\mathsf{LHS}_k} \Longleftrightarrow V_{\mathsf{NAC}}^{sh} \neq \emptyset \wedge \forall x \in V_{\mathsf{NAC}}^{sh}, \exists c \in V_M : s_{\mathsf{LHS}_k}^m(x) = c \wedge s_{\mathsf{NAC}}^m(x) = c\Big)$.

Definition 65 Given a model M and snapshot trees $\mathcal{ST}_{\mathsf{LHS}}$ and $\mathcal{ST}_{\mathsf{NAC}}$ defined for pattern graphs LHS and NAC, respectively, by also using search plans defined for the corresponding adorned search graphs, a **snapshot** s_{NAC} **invalidates a subtree rooted at snapshot** s_{LHS_k} (denoted by $s_{\mathsf{NAC}} \rightarrowtail \langle s_{\mathsf{LHS}_k} \rangle$), if snapshot s_{LHS_k} is invalidated by snapshot s_{NAC}, but its parent $p(s_{\mathsf{LHS}_k})$ is not invalidated by the same snapshot s_{NAC}. Formally, $\forall s_{\mathsf{LHS}_k} \in \mathcal{S}^{\mathcal{T}}_{\mathsf{LHS}}, \forall s_{\mathsf{NAC}} \in \mathcal{S}^{\mathcal{T}}_{\mathsf{NAC}}$: $(s_{\mathsf{NAC}} \rightarrowtail \langle s_{\mathsf{LHS}_k} \rangle \Longleftrightarrow s_{\mathsf{NAC}} \rightarrowtail s_{\mathsf{LHS}_k} \wedge \neg (s_{\mathsf{NAC}} \rightarrowtail p(s_{\mathsf{LHS}_k})))$.

Definition 66 Given a model M and snapshot trees $\mathcal{ST}_{\mathsf{LHS}}$ and $\{ \mathcal{ST}_{\mathsf{NAC}_i} \}$ defined for pattern graphs LHS and $\{ \mathsf{NAC}_i \}$, respectively, by also using search plans defined for the corresponding adorned search graphs, **invalidation edges** $\mathcal{I}$ denote the set of all such snapshot pairs $(s_{\mathsf{NAC}_i}, s_{\mathsf{LHS}_k})$, where the first snapshot s_{NAC_i} invalidates the subtree rooted at snapshot s_{LHS_k}. Formally, $\mathcal{I} \subseteq \bigcup_i \mathcal{S}^{\mathcal{T}}_{\mathsf{NAC}_i} \times \mathcal{S}^{\mathcal{T}}_{\mathsf{LHS}}$, and $\forall i, \forall s_{\mathsf{NAC}_i} \in \mathcal{S}^{\mathcal{T}}_{\mathsf{NAC}_i}, \forall s_{\mathsf{LHS}_k} \in \mathcal{S}^{\mathcal{T}}_{\mathsf{LHS}}$: $(s_{\mathsf{NAC}_i}, s_{\mathsf{LHS}_k}) \in \mathcal{I} \Longleftrightarrow s_{\mathsf{NAC}_i} \rightarrowtail \langle s_{\mathsf{LHS}_k} \rangle$.

8.3.4 Notification arrays

Since the transformation engine sends notifications on model changes, notification related data structures (shown in Fig. 8.3(b)) are also needed. The incremental pattern matching engine has a single *insert notification array* and a single *delete notification array* consisting of notification entries.

- An entry in the insert notification array is a pair consisting of an insert key and a list of snapshots to be notified. An *insert key* denotes a trigger condition for initiating incremental algorithms when objects or links are inserted into the model.
- An entry in the insert notification array is a pair consisting of a delete key and a list of snapshots to be notified. A *delete key* denotes a trigger condition for initiating incremental algorithms when objects or links are removed from the model.

The exact role of insert and delete notification arrays is presented later by Algorithm 8.1.

Example 34 Sample notification arrays are presented e.g., in the left part of Fig. 8.4(d). The `INSERT` notification array has 4 entries of which the first is triggered by the insert key `[*,type,Class]` and refers to snapshot `1`. This entry means that snapshot `1` has to be notified, when an object, which conforms to class Class is inserted into the model. Similarly, the first entry in the `DELETE` notification array means that snapshot `3` must be notified, if object `c1`, which conforms to class Class is deleted.

Definition 67 Given a metamodel MM and a model M, **insert keys** $\mathcal{K}_I$ denote trigger conditions for initiating incremental algorithms when objects or links are inserted into model M. Insert keys can be partitioned into three types (namely, $\mathcal{K}_I^1$, $\mathcal{K}_I^2$, and $\mathcal{K}_I^3$). Formally, $\mathcal{K}_I = \mathcal{K}_I^1 \cup \mathcal{K}_I^2 \cup \mathcal{K}_I^3$, and $\mathcal{K}_I^1 \cap \mathcal{K}_I^2 \cap \mathcal{K}_I^3 = \emptyset$.

- An insert key $\left[* \overset{\texttt{type}}{\rightarrow} C \right]$ of type $\mathcal{K}_I^1$ is triggered, if an object that conforms to class C is inserted into model M. Formally, $\mathcal{K}_I^1 \subseteq (\{ * \} \times \{ \texttt{type} \} \times V_{MM})$.
- An insert key $\left[* \overset{A}{\rightarrow} b \right]$ of type $\mathcal{K}_I^2$ is triggered, if a link of type A leading into object b is inserted into model M. Formally, $\mathcal{K}_I^2 \subseteq (\{ * \} \times Assoc \times V_M)$.
- An insert key $\left[a \overset{A}{\rightarrow} * \right]$ of type $\mathcal{K}_I^3$ is triggered, if a link of type A leading out of object a is inserted into model M. Formally, $\mathcal{K}_I^3 \subseteq (V_M \times Assoc \times \{ * \})$.

Definition 68 The **insert notification array** $\texttt{INSERT} : \mathcal{K}_I \rightarrow 2^{\mathcal{S}}$ maps each insert key to the set of such snapshots, which have to be processed by incremental algorithms.

Definition 69 Given a metamodel MM and a model M, **delete keys** $\mathcal{K}_D$ denote trigger conditions for initiating incremental algorithms when objects or links are deleted from model M. Delete keys can be partitioned into two types (namely, $\mathcal{K}_D^1$, $\mathcal{K}_D^2$). Formally, $\mathcal{K}_D = \mathcal{K}_D^1 \cup \mathcal{K}_D^2$, and $\mathcal{K}_D^1 \cap \mathcal{K}_D^2 = \emptyset$.

- A delete key $\left[c \overset{\texttt{type}}{\rightarrow} C\right]$ of type $\mathcal{K}_D^1$ is triggered, if an object c whose direct type is a descendant of class C is deleted from model M. Formally, $\mathcal{K}_D^1 \subseteq (V_M \times \{\,\texttt{type}\,\} \times V_{MM})$.
- A delete key $\left[a \overset{A}{\rightarrow} b\right]$ of type $\mathcal{K}_D^2$ is triggered, if a link of type A connecting object a to object b is deleted from model M. Formally, $\mathcal{K}_D^2 \subseteq (V_M \times Assoc \times V_M)$.

Definition 70 The **delete notification array** $\texttt{DELETE} : \mathcal{K}_D \rightarrow 2^{\mathcal{S}}$ maps each delete key to the set of such snapshots, which have to be modified by incremental algorithms.

8.3.5 Query results.

A *result set* (not shown in figures) is also maintained for each LHS pattern to speed-up the queries of complete matchings initiated by the GT tool that use the services of the incremental pattern matching approach.

Definition 71 Given a model M and a snapshot tree $\mathcal{ST}_{\mathsf{LHS}}$ defined for the LHS by also using the search plan SP defined for the corresponding adorned search graph ASG, the **result set** R_{LHS} **for the** LHS **graph** consists of such snapshots of tree $\mathcal{ST}_{\mathsf{LHS}}$, which are located on its deepest possible level without being invalidated by any snapshots of any NACs. Formally, $R_{\mathsf{LHS}} \subseteq \mathcal{S}_{\mathsf{LHS}}^{\mathcal{T}}$, and

$$\forall s_{\mathsf{LHS}_k} \in \mathcal{S}_{\mathsf{LHS}}^{\mathcal{T}} : \left(s_{\mathsf{LHS}_k} \in R_{\mathsf{LHS}} \Longleftrightarrow \mathsf{LHS}_k = \mathsf{LHS} \wedge \forall s_{\mathsf{NAC}_i} \in \bigcup_i \mathcal{S}_{\mathsf{NAC}_i}^{\mathcal{T}} : (s_{\mathsf{NAC}_i}, s_{\mathsf{LHS}_k}) \notin \mathcal{I} \right).$$

8.4 Operations for incremental pattern matching

During the incremental operation phase, the snapshot tree is maintained by four main methods.

(a) The `insert()` method (defined later by Algorithm 8.2) is responsible for the possible extension of the current matching for proper subpattern G_k to create a new matching for one larger subpattern G_{k+1}.

(b) The `validate()` method (defined later by Algorithm 8.3) is responsible for the recursive extension of insert operations to all (larger) subpatterns.

(c) The `delete()` method (defined later by Algorithm B.1) removes the whole matching subtree rooted at the current snapshot for subpattern G_k.

(d) The `invalidate()` method (defined later by Algorithm B.2) is responsible for the recursive deletion of all children snapshots of the current snapshot.

These methods are called by the pattern matching engine (see Algorithm 8.1) when modification events arrive from the model repository.

Algorithm 8.1 The core algorithm of the incremental pattern matching engine

1: **for all** $a \xrightarrow{e} b \in \Delta E_M^-$ **do**
2: **for all** $s \in \texttt{DELETE}\left(\left[a \xrightarrow{t(e)} b\right]\right)$ **do**
3: $\texttt{delete}(s)$
4: **end for**
5: **end for**
6: **for all** $c \in \Delta V_M^-$ **do**
7: **for all** $\left\{ C \in V_{MM} \mid C \xleftarrow{*} t(c) \right\}$ **do**
8: **for all** $s \in \texttt{DELETE}\left(\left[c \xrightarrow{\texttt{type}} C\right]\right)$ **do**
9: $\texttt{delete}(s)$
10: **end for**
11: **end for**
12: **end for**
13: **for all** $c \in \Delta V_M^+$ **do**
14: **for all** $\left\{ C \in V_{MM} \mid C \xleftarrow{*} t(c) \right\}$ **do**
15: **for all** $s \in \texttt{INSERT}\left(\left[* \xrightarrow{\texttt{type}} C\right]\right)$ **do**
16: $\texttt{insert}(s, c)$
17: **end for**
18: **end for**
19: **end for**
20: **for all** $a \xrightarrow{e} b \in \Delta E_M^+$ **do**
21: **for all** $s \in \texttt{INSERT}\left(\left[a \xrightarrow{t(e)} *\right]\right)$ **do**
22: $\texttt{insert}(s, b)$
23: **end for**
24: **for all** $s \in \texttt{INSERT}\left(\left[* \xrightarrow{t(e)} b\right]\right)$ **do**
25: $\texttt{insert}(s, a)$
26: **end for**
27: **end for**

- **Delete notification.** If a link $a \xrightarrow{e} b$ of type $t(e)$ connecting object a to b is removed from the model, then the `delete()` method is invoked with every snapshot being notified by the entry DELETE $\left(\left[a \xrightarrow{t(e)} b\right]\right)$ (lines 1–5). If an object c of type $t(c)$ is removed from the model, then the `delete()` method is invoked with every snapshot being notified by the entries DELETE $\left(\left[c \xrightarrow{\texttt{type}} C\right]\right)$, in which C iterates over all ancestors of $t(c)$ (lines 6–12).

- **Insert notification.** If an object c of type $t(c)$ is added to the model, then for all ancestors C of type $t(c)$ the `insert()` method is invoked with object c as a second parameter, and all snapshots being notified by the entries INSERT $\left(\left[* \xrightarrow{\texttt{type}} C\right]\right)$ as first parameter (lines 13–19). If a link $a \xrightarrow{e} b$ of type $t(e)$ connecting object a to b is added to the model, then the `insert()` method is invoked with every matching defined by entry INSERT $\left(\left[a \xrightarrow{t(e)} *\right]\right)$ and with object b as its parameters (lines 21–23). Then the same method is invoked with every matching defined by entry INSERT $\left(\left[* \xrightarrow{t(e)} b\right]\right)$ and with object a as its parameters (lines 24–26).

8.4.1 Incremental operations on an example

Prior to the detailed discussion of the algorithms, we first exemplify the process by using our running example of Fig. 8.4. Let us suppose that a class c1 is added to package p in the model by user interaction initiated by the system designer. The pattern matching engine is notified about this activity in two steps. First a notification arrives about the insertion of an object c1 of type Class (see Fig. 8.4(c)) followed by the insertion of an EO link connecting c1 to p (see Fig. 8.4(e)). Modifications are denoted by thick lines.

Step 1. When the object c1 of type Class is inserted, the pattern matching engine looks up entries retrieved by insert keys `[*,type,ModelElement]`, `[*,type,Namespace]`, and `[*,type,Class]`.

The last entry triggers the possible extension of snapshot 1 by mapping pattern node C to object c1 by invoking the `insert()` method with snapshot 1, and object c1 as parameters. As this is a matching for subpattern LHS_1, a new snapshot 3 is created and added to the (snapshot) tree as a child of snapshot 1, and the mapping C to c1 is recorded.

Then snapshot 3 is inserted into the delete notification array with delete key `[c1,type,Class]`. This means that whenever object c1, which conforms to class Class (i.e., the object that has been just added) is removed, this snapshot should be deleted.

Effects of adding a new snapshot to the tree are recursively extended to find matchings for larger subpatterns by calling `validate()`. Snapshot 3 can be further extended (as shown by corresponding new entries being added to the insert notification array pointing to snapshot 3), whenever an EO link leading out of c1 or a new Package is added to the model *in the future*.

As also the *current content* of the model may extend snapshot 3, we initiate the possible extensions of this matching by checking the existence of at least the EO links leading out of object c1.[2] As no such links exist in our example, the algorithm terminates with the snapshot tree presented in Fig. 8.4(d).

[2]Note that the insert key generation and the possible further extension of snapshot 3 are guided by the one larger subpattern LHS_2.

Step 2. When the EO link connecting c1 to p is inserted (as shown by the thick line of Fig. 8.4(e)), snapshot 3 is first extended to a new snapshot 4 by mapping pattern node P to object p and by executing a sequence of `insert()` and `validate()` method calls as shown in Fig. 8.5.

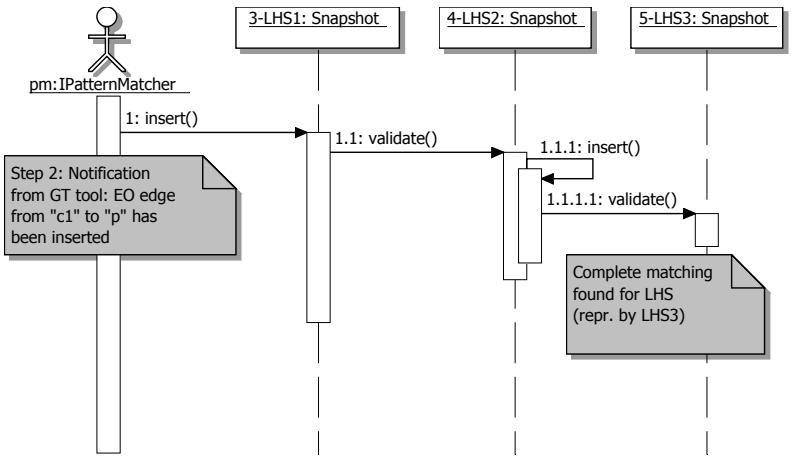


Figure 8.5: Sequence diagram showing edge insertion into the LHS pattern

This time, matching extension is propagated to another new snapshot 5 by assigning pattern node S to object s by invoking the `insert()` method with snapshot 4 and object s as parameters, as the current model already contained a schema s and a Ref link connecting p to s.

In addition, both new snapshots are appropriately registered in both the insert and delete notification arrays, and the binding array is updated accordingly. The corresponding snapshot tree is shown in Fig. 8.4(f).

At this point, snapshot 5 represents a (complete) matching for the LHS pattern, so the GT rule ClassRule can be applied.

Step 3. The result of applying ClassRule on the matching represented by snapshot 5 can be observed in Fig. 8.4(g) after the insertion of 3 objects and 5 links, processed one by one by the pattern matching engine.

Let us suppose that table t1 is inserted first. At this point, the insert notification array is consulted by retrieving snapshots that can be found at locations `[*,type,ModelElement]`, `[*,type,Namespace]`, `[*,type,Class]`, and `[*,type,Table]`. The latter two returns snapshots 1 and 2, respectively, so the `insert()` method is invoked with these snapshots and table t1 as input parameters. When snapshot 1 is processed, nothing changes in the snapshot tree. On the other hand, when snapshot 2 is trying to be extended, then snapshot 6 is created as a child.

In the following step, snapshot 7 is added as a child of snapshot 6, if the Ref link connecting class c1 to table t1 is inserted next into the model. As snapshot 7 represents a (complete) matching for the NAC pattern, snapshot 3 must be invalidated by deleting all its descendant snapshots in the tree. When all the new elements are added, the data structure will reflect the situation in Fig. 8.4(h).

8.4.2 Insert method

The insert method (shown by Algorithm 8.2) is responsible for the possible extension of the current partial matching for proper subpattern G_k to compute a new partial matching for subpattern G_{k+1}. If the current snapshot represents a complete matching for pattern G, then the method immediately terminates as matchings for pattern G can never be further extended.

Algorithm 8.2 The snapshot insertion algorithm `insert`(s_{G_k}, c)

PROCEDURE `insert`(s_{G_k}, c)
1: **if** $G_k \subset G$ **then**
2: {If $s^m_{G_k}$ is a matching for a proper subpattern G_k, and, thus, it is *not* a matching for pattern G}
3: **if** `checkGraphMorphism`(s_{G_k}, c) **then**
4: {If matching $s^m_{G_k}$ can be successfully extended by mapping the (k+1)th pattern node v_{k+1} to object c}
5: $s_{G_{k+1}}$:= `copyMatchings`(s_{G_k}, c) {Copy current matchings to the new matching}
6: `addDeleteEntries`$(s_{G_{k+1}})$ {New delete entries for matchings of condition edges}
7: **if** $\nexists s_{\mathsf{NAC}} \in \bigcup_i \mathcal{S}^{\mathcal{J}}_{\mathsf{NAC}_i} : (s_{\mathsf{NAC}}, s_{G_{k+1}}) \in \mathcal{J}$ **then**
8: `validate`$(s_{G_{k+1}})$ {Extend the new matching if not invalidated by any NACs}
9: **end if**
10: **end if**
11: **end if**

- The insert method is invoked with the current snapshot s_{G_k} and an object c, which is supposed to be the mapping of the (k+1)th pattern node in a new *potential matching*, which also contains all mappings defined by the matching for subpattern G_k being represented by the current snapshot.
- Since the current snapshot already represents a matching for the kth subpattern G_k, only mappings of the (k+1)th pattern node and its incoming and outgoing condition edges, which have just been defined by the new potential matching, are required to be checked by the `checkGraphMorphism()` method.
- If the potential matching is a correct graph morphism (and the `checkGraphMorphism()` returns true), the potential matching can be considered as a new matching for subpattern G_{k+1}. As such, a new snapshot is created and inserted into the snapshot tree by invoking the `copyMatchings()` method with the current snapshot s_{G_k} and object c as input parameters.
- The new snapshot is added to the delete notification array at all locations defined by the mappings of the (k+1)th pattern node and its incoming and outgoing condition edges.
- If the new snapshot is being invalidated by any (complete) matchings of any NAC patterns, then the `insert()` method terminates.
- Otherwise, the `validate()` method is invoked on the new snapshot trying to recursively extend the matching it represents.

8.4.3 Validate method

The validate method (shown by Algorithm 8.3) is responsible for the recursive extension of insert operations. It is invoked either when a new snapshot has been inserted into the snapshot tree and its

further extensions have to be checked (see Algorithm 8.2), or when extensions of the current matching possibly become valid due to the removal of a (complete) matching for an embedded NAC pattern (by the `invalidate()` method).

Algorithm 8.3 The snapshot validation algorithm `validate`(s_{G_k})

PROCEDURE `validate`(s_{G_k})
1: **if** $G_k = G$ **then**
2: {If $s^m_{G_k}$ is a *(complete) matching* for pattern G}
3: **if** $G =$ LHS **then**
4: {If $s^m_{G_k}$ is a (complete) matching for an LHS pattern}
5: $R'_{\mathsf{LHS}} := R_{\mathsf{LHS}} \cup \{\, s_{G_k} \,\}$ {Add the snapshot to the results}
6: **else**
7: {If $s^m_{G_k}$ is a (complete) matching for a NAC pattern}
8: **for all** $\left\{\, s \in \mathbb{S}^{\mathcal{T}}_{\mathsf{LHS}} \mid s_{G_k} \rightarrowtail \langle s \rangle \wedge (s_{G_k}, s) \notin \mathcal{I} \,\right\}$ **do**
9: **if** $\forall s_{\mathsf{NAC}} \in \bigcup_i \mathbb{S}^{\mathcal{T}}_{\mathsf{NAC}_i} : (s_{\mathsf{NAC}}, s) \notin \mathcal{I}$ **then**
10: `invalidate`(s) {If snapshot s has not been invalidated yet by any other snapshots s_{NAC}, then invalidate s.}
11: **end if**
12: $\mathcal{I}' := \mathcal{I} \cup \{\, (s_{G_k}, s) \,\}$
13: **end for**
14: **end if**
15: **else**
16: {If $s^m_{G_k}$ is a *partial* matching for pattern G}
17: `addInsertEntries`(s_{G_k}) {Add insert entries}
18: `propagateInsert`(s_{G_k}) {Propagate it to find a mapping of the next pattern node}
19: **end if**

- If snapshot s_{G_k} represents a (complete) matching for a LHS pattern, then the current snapshot is inserted into the result set R_{LHS}.
- If snapshot s_{G_k} represents a complete matching for a NAC pattern, then all snapshots s of the LHS pattern whose partial matching maps the shared pattern nodes to the same objects as the current matching have to be invalidated, if they have not been invalidated yet.
- By invoking the `addInsertEntries()` method, the current snapshot s_{G_k} is added to the insert notification array at locations defined by the *one larger* subpattern G_{k+1}. In this sense, the snapshot is inserted at locations $\left[* \overset{\texttt{type}}{\rightarrow} t(v_{k+1})\right]$, $\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} *\right]$, and $\left[* \overset{t(z)}{\rightarrow} s^m_{G_k}(w))\right]$ defined by the (k+1)th pattern node v_{k+1} and its incoming and outgoing condition edges $u \overset{z}{\rightarrow} v_{k+1}$ and $v_{k+1} \overset{z}{\rightarrow} w$, respectively.
- In the `propagateInsert()` method, insertion is attempted to be propagated to a matching for the one larger subpattern G_{k+1}. In this sense, an arbitrary incoming or outgoing condition edge of the (k+1)th pattern node is selected from subpattern G_{k+1}. If an incoming condition edge has been chosen, then we lookup all type conformant links leading out of the matched source object $s^m_{G_k}(u)$ of condition edge $u \overset{z}{\rightarrow} v_{k+1}$ and try to extend the current matching by mapping

the (k+1)th pattern node v_{k+1} to the target object of all iterated links, which is represented by the invocation of the `insert()` method with the current snapshot s_{G_k} and target object b as input parameters. Similarly, if an outgoing condition edge is selected, then we lookup all type conformant links leading into the matched target object $s^m_{G_k}(w)$ of condition edge $v_{k+1} \xrightarrow{z} w$ and try to extend the current matching by mapping the (k+1)th pattern node v_{k+1} to the source object of all iterated links, which is represented by the invocation of the `insert()` method with the current snapshot s_{G_k} and source object a as parameters. If the (k+1)th pattern node v_{k+1} is unconnected, then the `insert()` method is invoked with the current snapshot s_{G_k}, and all objects c in the model that conform to the type $t(v_{k+1})$ of pattern node v_{k+1}.

8.4.4 Delete and invalidate methods

Delete and invalidate methods implement the inverse operation of insert and validate methods, respectively.

The `delete()` method removes the whole subtree rooted at the current snapshot by removing all snapshots of the subtree from the notification arrays and the result set, and erasing all "dangling" invalidation links.

The `invalidate()` method deletes the whole subtree *excluding* the current snapshot, thus, it starts recursive deletion at its children. Another difference is that in case of validate method, the current snapshot is only removed from the insert notification array, and it remains in the delete notification array.

If the current snapshot represents a (complete) matching of an LHS then the `invalidate()` method removes the snapshot from the result set. If it represents a (complete) matching of a NAC pattern, then it (re)validates all snapshots invalidated previously by the current snapshot. On the implementation level, delete and invalidate methods mutually invoke each other, while descending in the tree for recursive matching removal. Algorithms for the `delete()` and `invalidate()` methods (as well as all auxiliary algorithms) are listed in Appendix B.

8.5 Experimental Evaluation

In order to assess the performance of our incremental approach, we performed measurements on the object-relational mapping benchmark example already introduced in Sec. 5.4. As a reference for the measurements, we selected Fujaba [44] as it is among the fastest non-incremental GT tools.

Only the test case in which rules are executed sequentially is used in the measurements. The parameter N was fixed to 10, 30, 50 and 100 for the runs.

Measurements were performed on a 1500 MHz Pentium machine with 768 MB RAM. A Linux kernel of version 2.6.7 served as an underlying operating system. The execution time results are shown in Table 8.1. Note that this table only contains measurement results for the operation phase of the incremental pattern matcher. Preprocessing and initialization phases are not analyzed quantitatively as they only run once, and our aim in this experimental evaluation was to compare the regular behaviour of non-incremental and incremental graph transformation.

The head of a row shows the name of the rule on which the average is calculated. (Note that a rule is executed several times in a run.) The second column (Class) depicts the number of classes in the run, which is, in turn, the runtime parameter N for the test case. The third and fourth columns show the concrete values for the model size (meaning the number of objects and links in the model) and the transformation sequence length, respectively. Heads of the remaining columns unambiguously identify the approach having been used. Values in match and update columns depict the average times needed for

	Class	Model	TS	Fujaba		Incremental	
		size	length	match	update	match	update
	#	#	#	msec	msec	msec	msec
ClassRule	10	1342	146	0.201	0.479	0.026	5.439
	30	12422	1336	0.287	0.052	0.023	56.116
	50	34702	3726	0.171	0.012	0.021	221.955
	100	139402	14951	0.278	0.011	0.042	2067.462
AssocRule	10	1342	146	0.937	0.148	0.019	1.665
	30	12422	1336	2.488	0.101	0.032	4.510
	50	34702	3726	3.371	0.032	0.022	6.849
	100	139402	14951	11.959	0.030	0.039	26.684
AssocEndRule	10	1342	146	0.875	0.107	0.043	0.592
	30	12422	1336	3.896	0.045	0.016	1.108
	50	34702	3726	5.975	0.025	0.023	1.948
	100	139402	14951	24.057	0.028	0.068	9.353

Table 8.1: Experimental results

a single execution of a rule in the pattern matching and updating phase, respectively. Execution times were measured on a microsecond scale, but a millisecond scale is used in Table 8.1 for presentation purposes.

Our experiments can be summarized as follows.

- In accordance with our assumptions, the incremental engine executes pattern matching in constant time even in case of large models, while the traditional engine shows significant increase when the LHS of the pattern is large as in case of AssocEndRule.
- Incremental techniques by their nature suffer time increase in the updating phase due to the bookkeeping overhead caused by the additional data structures, and the fact that even the insertion of a single edge may generate (or delete) a significant amount of matchings. Its detrimental performance effects are reported in the updating phase of ClassRule, when also the matchings of the other rules have to be refreshed. On the other hand, the traditional engine executes the update phase in constant time as it can be expected.
- By taking into account both phases in the analysis, it may be stated that the incremental strategy provides a competitive alternative for traditional engines as the total execution times of the incremental approach are of the same order of magnitude in case of the frequently applied rules (i.e., AssociationRule and AssocEndRule).
- The benefits of the incremental approach are the most remarkable when rules have complex LHS graphs as the pattern matching of Fujaba gets slow in this case and when the dependency between rules is weak as this leads to fast updates in incremental engines.

As a consequence, we may draw that the incremental approach is a primary candidate for such application domains of graph transformation where complex transformation rules are used and where

all matchings of a rule have to be accessed rapidly, which is a typical case for analysis/verification tools. In addition, incremental graph transformation might be applicable to model-to-model translations where the synchronization of models has to be ensured, and to rule-based pattern recognition for image processing purposes [16].

8.6 Incremental graph pattern matching in relational databases

Chapter 6 proposed an approach for representing graph transformation in relational databases. In this approach, objects and links of the instance model have been persistently stored in database tables as presented in Sec. 6.4.1. Graph pattern matching has been defined by queries specifying one database view for each LHS pattern, NAC pattern, and rule precondition as discussed in Sections 6.4.2 and 6.4.3. The updating phase has been implemented by executing data manipulation commands, which modify the tables that store the instance model as shown in Sec. 6.4.4.

Now I present how incremental techniques can be used when graph transformation is defined on top of a relational database. The goal is still to avoid the complete recalculation of matchings, which can be achieved by persistently storing matchings in views, and by tracking how modifications in tables representing objects and links of the instance model are propagated as changes in the views expressing LHS patterns, NAC patterns and rule preconditions.

Since the proposed RDBMS based incremental technique uses the definitions and algorithms of the non-incremental approach of Chapter 6, only concepts needed for the incremental behaviour are now discussed.

8.6.1 Events and triggers

An incremental approach operating in relational databases has to be able to detect modifications in tables and views, and to handle these changes by propagating modifications to other views.

- Tables and views report on changes in their content in the form of *events*, which identify the set of *modified rows*, and contain an *event type descriptor* for marking the kind of modification. As the content of tables and views can be altered by adding rows, by removing rows and also by modifying values in rows, these changes are reported by *insertion*, *deletion*, and *update events*, respectively. Events can be *filtered* by a formula by keeping only those modified rows, which fulfill the formula that specifies the filtering condition.

- Change propagation is controlled by *triggers*, which express compile-time dependencies of views on tables or on other views. The *condition* part of a trigger indicates when its *action* part has to be executed. A trigger condition is typically specified by an event issued by a table or view, while the action part describes modifications in the content of the depending view by means of queries. As this approach only allows for having row additions and removals in the action part, corresponding triggers are referred as *insert* and *delete operation triggers*, respectively.

Definition 72 Given a database table $\mathcal{T}(A_1, \ldots, A_n)$ with n columns, on which a data manipulation (insert, delete, or update) operation has been executed resulting in a table content $\mathcal{T}'$, an **event issued by table** $\mathcal{T}$ (denoted by $(\mathrm{e}, \Delta\mathcal{T})$) is a notification being sent when the content of table $\mathcal{T}$ is changed, and it consists of an **event type descriptor** from the set $\{$ `INSERT`, `DELETE`, `UPDATE` $\}$ identifying the reason for the modification, and a set of **modified rows** (denoted by $\Delta\mathcal{T}$). Modified rows always constitute a subset of those rows that might be stored in table $\mathcal{T}$. Formally, $\Delta\mathcal{T} \subseteq (C_1 \cup \{\varepsilon\}) \times \ldots \times (C_n \cup \{\varepsilon\})$, where C_i denotes the set of values allowed in the ith column A_i of table $\mathcal{T}$ (see Definition 29).

Based on the type of data manipulation operation performed, events can be partitioned into the following three groups.

- An **insertion event** $(\texttt{INSERT}, \Delta\mathcal{T}^+)$ is issued, whenever rows $\Delta\mathcal{T}^+$ are added to table $\mathcal{T}$ resulting in a content $\mathcal{T}'$. Formally, $\Delta\mathcal{T}^+ = \mathcal{T}' \setminus \mathcal{T}$, and $(\texttt{INSERT}, \Delta\mathcal{T}^+) \subseteq \mathcal{E}_{\mathcal{T}}^{\texttt{INSERT}}$, where
$$\mathcal{E}_{\mathcal{T}}^{\texttt{INSERT}} = \{\, \texttt{INSERT} \,\} \times (C_1 \cup \{\, \varepsilon \,\}) \times \ldots \times (C_n \cup \{\, \varepsilon \,\}) \,.$$
- A **deletion event** $(\texttt{DELETE}, \Delta\mathcal{T}^-)$ is issued, whenever rows $\Delta\mathcal{T}^-$ are removed from database table $\mathcal{T}$ resulting in a content $\mathcal{T}'$. Formally, $\Delta\mathcal{T}^- = \mathcal{T} \setminus \mathcal{T}'$, and $(\texttt{DELETE}, \Delta\mathcal{T}^-) \subseteq \mathcal{E}_{\mathcal{T}}^{\texttt{DELETE}}$, where
$$\mathcal{E}_{\mathcal{T}}^{\texttt{DELETE}} = \{\, \texttt{DELETE} \,\} \times (C_1 \cup \{\, \varepsilon \,\}) \times \ldots \times (C_n \cup \{\, \varepsilon \,\}) \,.$$
- An **update event** $\left(\texttt{UPDATE}, \Delta\mathcal{T}_{new}^{A_i}\right)$ **on column** A_i is issued, whenever only the value in column A_i of rows $\Delta\mathcal{T}_{old}^{A_i}$ of table $\mathcal{T}$ are modified resulting in rows $\Delta\mathcal{T}_{new}^{A_i}$ in a table content $\mathcal{T}'$. Formally, $\Delta\mathcal{T}_{new}^{A_i} = \{\, \vec{z}_{new} \in \mathcal{T}' \mid \exists \vec{z}_{old} \in \mathcal{T},\, \forall k \in \mathbb{Z}_n^+ \,:\, (i \neq k \iff \vec{z}_{new}[A_k] = \vec{z}_{old}[A_k]) \,\}$, and $\left(\texttt{UPDATE}, \Delta\mathcal{T}_{new}^{A_i}\right) \subseteq \mathcal{E}_{\mathcal{T}}^{\texttt{UPDATE}}$, where
$$\mathcal{E}_{\mathcal{T}}^{\texttt{UPDATE}} = \{\, \texttt{UPDATE} \,\} \times (C_1 \cup \{\, \varepsilon \,\}) \times \ldots \times (C_n \cup \{\, \varepsilon \,\}) \,.$$

The set of all possible events can be described formally by $\mathcal{E}_{\mathcal{T}} = \mathcal{E}_{\mathcal{T}}^{\texttt{INSERT}} \cup \mathcal{E}_{\mathcal{T}}^{\texttt{DELETE}} \cup \mathcal{E}_{\mathcal{T}}^{\texttt{UPDATE}}$.

Definition 73 Given a formula F, an **event filtered by formula** F (denoted by $(\mathrm{e}, \sigma_F(\Delta\mathcal{T}))$) contains only such modified rows of event $(\mathrm{e}, \Delta\mathcal{T})$, for which $F(y_1, \ldots, y_n)$ holds. Formally,
$$(\mathrm{e}, \sigma_F(\Delta\mathcal{T})) = \{\, (\mathrm{e}, y_1, \ldots, y_n) \mid (\mathrm{e}, y_1, \ldots, y_n) \in (\mathrm{e}, \Delta\mathcal{T}) \wedge F(y_1, \ldots, y_n) = \text{ true} \,\} \,.$$
An obvious corollary is that $(\mathrm{e}, \sigma_F(\Delta\mathcal{T})) \subseteq (\mathrm{e}, \Delta\mathcal{T})$.

Definition 74 Given database tables $\mathcal{S}$ and $\mathcal{T}$, an **insert operation trigger specified by query** Q^+ **for table** $\mathcal{T}$ **on event** $(\mathrm{e}, \Delta\mathcal{S})$ **being issued by table** $\mathcal{S}$ (denoted by $\mathcal{S} \vdash (\mathrm{e}, \Delta\mathcal{S}) \,/\, Q^+ \rightarrow \mathcal{T}$) means that result rows of query operation Q^+ are calculated and *added to* table $\mathcal{T}$ whenever an event $(\mathrm{e}, \Delta\mathcal{S})$ arrives. Formally, $\mathcal{T}' = \mathcal{T} \cup Q^+$.

Definition 75 Given database tables $\mathcal{S}$ and $\mathcal{T}$, a **delete operation trigger specified by query** Q^- **for table** $\mathcal{T}$ **on event** $(\mathrm{e}, \Delta\mathcal{S})$ **being issued by table** $\mathcal{S}$ (denoted by $\mathcal{S} \vdash (\mathrm{e}, \Delta\mathcal{S}) \,/\, Q^- \rightarrow \mathcal{T}$) means that result rows of query operation Q^- are calculated and *removed from* table $\mathcal{T}$ whenever an event $(\mathrm{e}, \Delta\mathcal{S})$ arrives. Formally, $\mathcal{T}' = \mathcal{T} \setminus Q^-$.

In the following, notation $Q\,[\mathcal{T} \rightarrow \mathcal{U}]$ denotes the query Q, in which all the occurrences of table $\mathcal{T}$ have been replaced by table $\mathcal{U}$.

8.6.2 Incremental view updates for rule graphs (LHS and NAC)

Section 6.4.2 already described how views for rule graphs (LHS and NAC) could be generated from tables that stored objects and links of the instance model. Now I present the process that updates these views incrementally by using events being issued when the content of tables is changed.

For each pattern node and edge of each LHS graph, a pair consisting of an insert and a delete operation trigger is specified for the view defined for the LHS. These triggers process events issued by the table representing the pattern node and edge in turn.

Trigger conditions. Conditions of triggers are defined by the following guidelines.

- As objects are represented by individual rows in the corresponding tables as shown in Sec. 6.4.1, the addition and the removal of such objects appear as insertion and deletion events issued by these tables, respectively.

  To handle these events during incremental operation, an insert and a delete operation trigger have to be declared, respectively, for each pattern node of each LHS (NAC) graph. These triggers propagate changes in the table representing the type of the pattern node to the view being assigned to the LHS (NAC) graph.

- Each many-to-one link is stored in the table that corresponds to its source object, and represented by the identifier of its target object in the column that has been assigned to the type of the link. In this case, modifications of such links are reported by update events, which inform on changes in the column associated with the link type. However, as link addition and removal are described by setting and clearing the corresponding target object identifiers, respectively, update events have to be filtered accordingly.

  In incremental mode, for each many-to-one link of each LHS (NAC) graph, an insert and a delete operation trigger are defined to process filtered update events, which report on setting and clearing values, respectively, in the column representing the link type.

- A many-to-many link is represented by an individual row in a corresponding table, the addition and the removal of such links are reported by insertion and deletion events, respectively. These events can be handled by a corresponding insert and delete operation trigger, respectively, which are defined for each many-to-many pattern edge of each LHS (NAC) graph.

Trigger actions. Recall that the non-incremental approach of Sec. 6.4.2 used a separate view for calculating the matchings of each LHS and NAC graph. Each view has been specified by a query consisting of the following sequence of operations. An inner join operation is executed first on tables that represent either a node or an edge of the rule graph. The joined table is then filtered by injectivity and edge constraints, and finally, columns that represent node identifiers are selected by a projection.

In the incremental approach, the above-mentioned overall query structure is preserved for defining trigger actions, only the table, which issues the events processed by the trigger, is replaced with the modified rows of the same event.

Example 35 For exemplifying the incremental operation in RDBMSs, the database equivalent of ClassRule is monitored, when database representatives of class c1 and EO link connecting class c1 to package p are added to the model as depicted by Figures 8.4(c), and 8.4(e), respectively.

As the LHS of ClassRule has three nodes and two many-to-one edges, there are five insert and five delete operation triggers defined for view ClassRule_lhs. Each pair of triggers handles events arriving from the table that represents the type of either a pattern node or a pattern edge.

This running example focuses only on those triggers that are actually used during the above-mentioned transformation. In this sense, when class c1 is added to the instance model, the insert operation trigger, which handles insertion events of table Class is invoked. The trigger action is described by the query of Listing 8.1.

During the actual query execution, the question mark in Listing 8.1 has to be substituted with the database identifier of the class being inserted, which is c1 in this case. This query gives an empty result as the equation `c_anc.EO = p.id` fails due to the lack of EO links in the model of Fig. 8.4(c). As a consequence, view ClassRule_lhs is not extended now.

When the EO link connecting class c1 to package p is added to the model as shown by Fig. 8.4(e), table ModelElement is modified in its column EO by modifying NULL value to p. This change is detected

```
INSERT INTO ClassRule_lhs (c,p,s)            -- Insert operation
SELECT c.id AS c, p.id AS p, s.id AS s       -- Non-incremental query
FROM Class AS c, ModelElement AS c_anc,
     Package AS p, ModelElement AS p_anc,
     Schema AS s
WHERE c.id = c_anc.id AND c_anc.EO = p.id
  AND p.id = p_anc.id AND p_anc.Ref = s.id
  AND p.id <> s.id
  AND c.id = ?                               -- Execution restricted to
                                             -- IDs of new classes
                                             -- in incremental mode
```

Listing 8.1: Query handling the insertion of classes

by the insert operation trigger defined for view ClassRule_lhs, which handles such events that signal an update on column EO, during which a NULL value has been replaced by a non-NULL value. The action defined by this insert operation trigger is depicted in Listing 8.2.

```
INSERT INTO ClassRule_lhs (c,p,s)
SELECT c.id AS c, p.id AS p, s.id AS s
FROM Class AS c, ModelElement AS c_anc,
     Package AS p, ModelElement AS p_anc,
     Schema AS s
WHERE c.id = c_anc.id AND c_anc.EO = p.id
  AND p.id = p_anc.id AND p_anc.Ref = s.id
  AND p.id <> s.id
  AND c_anc.id = ?  -- "c1"
  AND c_anc.EO = ?  -- "p"
```

Listing 8.2: Query handling the insertion of EO links

When the query of Listing 8.2 is executed, question marks are replaced with identifiers c1 and p denoting source and target objects of the new EO link, respectively. As a result, a triple consisting of values c1, p, and s is added to view ClassRule_lhs, which corresponds to a new matching for the LHS graph of ClassRule.

Formalization. Recall that notational shorthands $n_V = |V_{\mathsf{LHS}}|$ and $n_E = |E_{\mathsf{LHS}}|$ have been introduced for denoting the number of pattern nodes and edges. Additionally, a total order has been defined for the node and edge sets in which nodes precede edges, and x_i and z_{n_V+j} are the ith node and the jth edge in this order, respectively.

Recall further that the view r^d_{LHS} for the LHS has been defined in Sec. 6.4.2 by the following query

$$Q = \pi_{ProjColRefs}\left(\sigma_{Inj \wedge Edge}\left(\mathcal{T}_1 \times \ldots \times \mathcal{T}_i \times \ldots \times \mathcal{T}_{n_V+n_E}\right)\right),$$

in which table $\mathcal{T}_i$ has been assigned to the type of the ith pattern node or edge of r_{LHS}. Formally,

$$\mathcal{T}_i = \begin{cases} t(x_i)^d, \text{ when } i \leq n_V \text{ and } x_i \in V_{\mathsf{LHS}} \\ src(t(z_i))^d, \text{ when } n_V < i \leq n_V + n_E \text{ and } u_i \xrightarrow{z_i}_1 v_i \in E_{\mathsf{LHS}} \\ t(z_i)^d, \text{ when } n_V < i \leq n_V + n_E \text{ and } u_i \xrightarrow{z_i}_* v_i \in E_{\mathsf{LHS}} \end{cases}$$

Edge constraints $Edge$, injectivity constraints Inj, and projection column references $ProjColRefs$ are expressed in exactly the same way as in Sec. 6.4.2.

In order to incrementally update view r^d_{LHS} for rule graph r_{LHS}, insert and delete operation triggers are defined, which handle events arriving from tables representing the types of pattern nodes and edges. Queries of these triggers always use such versions of query Q, which restrict calculations only on the set of modified rows reported by arriving events instead of using the complete table.

- *Triggers for pattern nodes.* For each pattern node x_i, an insert operation trigger is specified by query $Q\left[t(x_i)^d \rightarrow \Delta(t(x_i)^d)^+\right]$ for view r^d_{LHS} on insertion event $\left(\texttt{INSERT}, \Delta(t(x_i)^d)^+\right)$ issued by the table $t(x_i)^d$ representing the type $t(x_i)$ of pattern node x_i.

  Additionally, for each pattern node x_i, a delete operation trigger is specified by query $Q\left[t(x_i)^d \rightarrow \Delta(t(x_i)^d)^-\right]$ for the same view r^d_{LHS} on deletion event $\left(\texttt{DELETE}, \Delta(t(x_i)^d)^-\right)$ issued by the same table $t(x_i)^d$.

- *Triggers for many-to-one pattern edges.* In order to avoid complex formulae when specifying triggers for many-to-one pattern edge $u_i \xrightarrow{z_i}_1 v_i$, let $\mathbb{S}_i$ temporarily denote the table $src(t(z_i))^d$, in which many-to-one links of type $t(z_i)$ are stored in the database. Since target objects of these links are represented by values in column $t(z_i)^d$, the database trigger monitoring changes in this column can be expressed by update events with modified rows $\Delta(\mathbb{S}_i)^{t(z_i)^d}_{new}$.

  For each many-to-one pattern edge $u_i \xrightarrow{z_i}_1 v_i$, an insert operation trigger is specified by query $Q\left[\mathbb{S}_i \rightarrow \sigma_{t(z_i)^d \neq \varepsilon}\left(\Delta(\mathbb{S}_i)^{t(z_i)^d}_{new}\right)\right]$ for view r^d_{LHS} on filtered update event $\left(\texttt{UPDATE}, \sigma_{t(z_i)^d \neq \varepsilon}\left(\Delta(\mathbb{S}_i)^{t(z_i)^d}_{new}\right)\right)$ issued by the table $\mathbb{S}_i$. By performing the selection $\sigma_{t(z_i)^d \neq \varepsilon}$ on update event $\left(\texttt{UPDATE}, \Delta(\mathbb{S}_i)^{t(z_i)^d}_{new}\right)$, only those modified rows are kept, in which the value in column $t(z_i)^d$ has been changed from undefined (NULL) to a non-NULL value.

  Additionally, for each many-to-one pattern edge $u_i \xrightarrow{z_i}_1 v_i$, a delete operation trigger is specified by query $Q\left[\mathbb{S}_i \rightarrow \sigma_{t(z_i)^d = \varepsilon}\left(\Delta(\mathbb{S}_i)^{t(z_i)^d}_{new}\right)\right]$ for the same view r^d_{LHS} on deletion event $\left(\texttt{UPDATE}, \sigma_{t(z_i)^d = \varepsilon}\left(\Delta(\mathbb{S}_i)^{t(z_i)^d}_{new}\right)\right)$ issued by the same table $\mathbb{S}_i$. In this case, the filtered event identifies such rows, in which the value in column $t(z_i)^d$ has been modified from a non-NULL value to undefined (NULL).

- *Triggers for many-to-many pattern edges.* For each many-to-many pattern edge $u_i \xrightarrow{z_i}_* v_i$, an insert operation trigger is specified by query $Q\left[t(z_i)^d \rightarrow \Delta(t(z_i)^d)^+\right]$ for view r^d_{LHS} on insertion event $\left(\texttt{INSERT}, \Delta(t(z_i)^d)^+\right)$ issued by the table $t(z_i)^d$ representing the type $t(z_i)$ of pattern edge $u_i \xrightarrow{z_i}_* v_i$.

  Additionally, for each many-to-many pattern edge $u_i \xrightarrow{z_i}_* v_i$, a delete operation trigger is specified by query $Q\left[t(z_i)^d \rightarrow \Delta(t(z_i)^d)^-\right]$ for the same view r^d_{LHS} on deletion event $\left(\texttt{DELETE}, \Delta(t(z_i)^d)^-\right)$ issued by the same table $t(z_i)^d$.

Insertion and deletion triggers for views representing the NACs can be specified in exactly the same way, but using the NAC graphs in the process.

8.6.3 Incremental updates for preconditions of rules

As it has been introduced in Sec. 6.2, the calculation of a view for the precondition of a rule proceeds as follows in the non-incremental case. Each NAC is left outer joined to the LHS graph one by one

by using join conditions, which express that columns representing the same shared node in different rule graphs should be equal. Additional filtering conditions require that columns of NAC(s), which are shared with the LHS part, have to be filled with undefined values. Finally, a projection displays only those columns that originate from LHS.

In the incremental approach, for each rule precondition view, a pair of insert and delete operation triggers are defined. Insertion and deletion events of the view for the LHS graph contained by the precondition in turn are handled by these triggers, respectively. Similarly, a pair of insert and delete operation triggers have to be specified for each precondition view for handling events arriving from each NAC view of the precondition. However, since new matchings for a NAC can narrow the set of solutions for the precondition, the dependency has to be transposed meaning that insert and delete operation triggers should handle deletion and insertion events, respectively.

The four possible cases of modifications are now discussed in details.

- **Augmenting** LHS **views.** When rows are added to the view defined for the LHS graph, the above-mentioned query has to be evaluated *on the new rows* resulting in a possible new set of rows to be inserted into the rule precondition view. This behaviour can be achieved by defining an insert operation trigger for the precondition view, which handles insertion events that arrive from the LHS view.

- **Narrowing** LHS **views.** When rows are removed from the LHS view, their counterparts in the precondition view also have to be removed, which is expressed by a delete operation trigger that handles deletion events of the LHS view.

- **Augmenting** NAC **views.** When a NAC view is augmented, then the region being blocked by these new rows has to be determined and removed from the precondition view. The *blocked region* consists of such rows of the LHS view, which can be inner joined to the new rows of the NAC view.

  This case can be handled by a delete operation trigger, which processes insertion events of the NAC view.

- **Narrowing** NAC **views.** When rows are removed from the NAC view, then the precondition view defining query has to be re-evaluated by substituting the LHS view with the region that has been blocked by such rows of the NAC view that have just been deleted. This is expressed by an insert operation trigger that handles deletion events of the NAC view.

Example 36 By continuing the previous running example, modifications in view ClassRule are now examined. When the database representation of class c1 has been added to table Class, the content of LHS view ClassRule_lhs is not changed. However, in the second round, when EO link connecting class c1 to package p is added, a new row appears in view ClassRule_lhs, which consists of identifiers c1, p, s, which represent a matching for the LHS graph in turn.

The appearance of this new row generates an insertion event issued by view ClassRule_lhs, which is processed by an insertion trigger of precondition view ClassRule, which executes the following query as an action.

In the above query, question marks are substituted with identifiers c1, p, and s, respectively, which originate from the new row, whose creation is reported by the insertion event. By evaluating the query, the same row is added to view ClassRule as well denoting a new matching for the precondition in turn.

```
CREATE VIEW ClassRule AS
SELECT lhs.*
FROM ClassRule_lhs AS lhs
  LEFT JOIN ClassRule_nac AS nac ON lhs.c = nac.c
WHERE nac.c IS NULL
  AND lhs.c = ?  -- "c1"
  AND lhs.p = ?  -- "p"
  AND lhs.s = ?  -- "s"
```

Listing 8.3: Query handling insertion events issued by view ClassRule_lhs

Formalization. Recall that the view r^d_{PRE}, which represents the precondition r_{PRE} consisting of a single LHS and k negative application conditions has been calculated by the following query in Sec. 6.4.3.

$$P = r^d_{\mathsf{PRE}} = \pi_{ProjColRefs}\left(\sigma_{Null}\left(r^d_{\mathsf{LHS}} \overset{F_1}{\ltimes} r^d_{\mathsf{NAC}_1} \overset{F_2}{\ltimes} \ldots \overset{F_k}{\ltimes} r^d_{\mathsf{NAC}_k}\right)\right)$$

In this query, column references $ProjColRefs$, null conditions $Null$, and join conditions F_i are the same as in Sec. 6.4.3. Recall also that view r^d_{PRE} constitutes a subset of view r^d_{LHS}.

Definition 76 Given the query $P = \pi_{ProjColRefs}\left(\sigma_{Null}\left(r^d_{\mathsf{LHS}} \overset{F_1}{\ltimes} r^d_{\mathsf{NAC}_1} \overset{F_2}{\ltimes} \ldots \overset{F_k}{\ltimes} r^d_{\mathsf{NAC}_k}\right)\right)$, which is used for calculating view r^d_{PRE}, a **region blocked by view** $r^d_{\mathsf{NAC}_i}$ contains such rows of view r^d_{LHS}, which should be invalidated by left joining view $r^d_{\mathsf{NAC}_i}$ by using formula F_i. The blocked region consists of such rows of view r^d_{LHS}, which can be successfully inner joined to view $r^d_{\mathsf{NAC}_i}$ by using filtering formula F_i. Formally,

$$R_{\mathsf{NAC}_i} = \pi_{ProjColRefs}\left(\sigma_{F_i}\left(r^d_{\mathsf{LHS}} \times r^d_{\mathsf{NAC}_i}\right)\right) \subseteq r^d_{\mathsf{LHS}}.$$

Based on the above definition and the structure of query P, a row might be invalidated by several NAC views $r^d_{\mathsf{NAC}_i}$, so blocked regions might overlap each other. On the other hand, blocked regions are always disjoint with view r^d_{PRE} as this latter contains exactly such rows of view r^d_{LHS} that have not been invalidated by any NAC views $r^d_{\mathsf{NAC}_i}$.

It is worth emphasizing that a NAC view $r^d_{\mathsf{NAC}_i}$ cannot influence which rows of view r^d_{LHS} to exclude from the result set outside its blocked region R_{NAC_i}. This observation is useful, when rows $\Delta\left(r^d_{\mathsf{NAC}_i}\right)^-$ are deleted from view $r^d_{\mathsf{NAC}_i}$ as they can only enable such rows of view r^d_{LHS} for possible re-inclusion, which have been previously blocked by the rows $\Delta\left(r^d_{\mathsf{NAC}_i}\right)^-$ to be deleted.

- *Triggers for tracking modifications in views representing LHS.* If rows $\Delta\left(r^d_{\mathsf{LHS}}\right)^+$ are added to view r^d_{LHS} as reported by the insertion event $\left(\texttt{INSERT}, \Delta\left(r^d_{\mathsf{LHS}}\right)^+\right)$, then query P has to be recomputed by using only the inserted rows $\Delta\left(r^d_{\mathsf{LHS}}\right)^+$ in the leftmost position of the left join operation instead of the complete view r^d_{LHS}. This can be expressed by

  $$P^+_{\mathsf{LHS}} = \pi_{ProjColRefs}\left(\sigma_{Null}\left(\Delta\left(r^d_{\mathsf{LHS}}\right)^+ \overset{F_1}{\ltimes} r^d_{\mathsf{NAC}_1} \overset{F_2}{\ltimes} \ldots \overset{F_k}{\ltimes} r^d_{\mathsf{NAC}_k}\right)\right).$$

  If rows $\Delta\left(r^d_{\mathsf{LHS}}\right)^-$ are removed from view r^d_{LHS} as reported by the deletion event $\left(\texttt{DELETE}, \Delta\left(r^d_{\mathsf{LHS}}\right)^-\right)$, then these rows must be removed from the result view r^d_{PRE} as well. The required changes can be expressed by query $P^-_{\mathsf{LHS}} = \Delta\left(r^d_{\mathsf{LHS}}\right)^-$.

For appropriately tracking the above-mentioned modifications of view r^d_{LHS}, an insert and a delete operation trigger should be specified by queries P^+_{LHS} and P^-_{LHS} for view r^d_{PRE} on insertion event $\left(\texttt{INSERT}, \Delta\left(r^d_{\mathsf{LHS}}\right)^+\right)$ and deletion event $\left(\texttt{DELETE}, \Delta\left(r^d_{\mathsf{LHS}}\right)^-\right)$, respectively.

- *Triggers for tracking modifications in views representing NAC.* If rows $\Delta\left(r^d_{\mathsf{NAC}_i}\right)^+$ are added to view $r^d_{\mathsf{NAC}_i}$ as reported by the insertion event $\left(\texttt{INSERT}, \Delta\left(r^d_{\mathsf{NAC}_i}\right)^+\right)$, then all such rows of view r^d_{LHS}, which can be successfully (inner) joined to new rows $\Delta\left(r^d_{\mathsf{NAC}_i}\right)^+$ of view $r^d_{\mathsf{NAC}_i}$, should be removed from view r^d_{PRE}, which can be expressed by the following query

$$P^-_{\mathsf{NAC}_i} = R_{\mathsf{NAC}_i}\left[r^d_{\mathsf{NAC}_i} \rightarrow \Delta\left(r^d_{\mathsf{NAC}_i}\right)^+\right] = \pi_{ProjColRefs}\left(\sigma_{F_i}\left(r^d_{\mathsf{LHS}} \times \Delta\left(r^d_{\mathsf{NAC}_i}\right)^+\right)\right).$$

If rows $\Delta\left(r^d_{\mathsf{NAC}_i}\right)^-$ are removed from view $r^d_{\mathsf{NAC}_i}$ as reported by the deletion event $\left(\texttt{DELETE}, \Delta\left(r^d_{\mathsf{NAC}_i}\right)^-\right)$, then first, those rows of view r^d_{LHS} have to be identified, which might reappear in view r^d_{PRE} due to the deletion of rows $\Delta\left(r^d_{\mathsf{NAC}_i}\right)^-$. These rows are in the region that has been blocked by deleted rows $\Delta\left(r^d_{\mathsf{NAC}_i}\right)^-$, which can formally be described by $R_{\mathsf{NAC}_i}\left[r^d_{\mathsf{NAC}_i} \rightarrow \Delta\left(r^d_{\mathsf{NAC}_i}\right)^-\right]$. Then query P has to be re-evaluated on the rows of the blocked region $R_{\mathsf{NAC}_i}\left[r^d_{\mathsf{NAC}_i} \rightarrow \Delta\left(r^d_{\mathsf{NAC}_i}\right)^-\right]$ to provide the set of rows from view r^d_{LHS} that have to be added to view r^d_{PRE}. Formally,

$$P^+_{\mathsf{NAC}_i} = \pi_{ProjColRefs}\left(\sigma_{Null}\left(R_{\mathsf{NAC}_i}\left[r^d_{\mathsf{NAC}_i} \rightarrow \Delta\left(r^d_{\mathsf{NAC}_i}\right)^-\right] \overset{F_1}{\ltimes} r^d_{\mathsf{NAC}_1} \overset{F_2}{\ltimes} \ldots \overset{F_k}{\ltimes} r^d_{\mathsf{NAC}_k}\right)\right),$$

where $R_{\mathsf{NAC}_i}\left[r^d_{\mathsf{NAC}_i} \rightarrow \Delta\left(r^d_{\mathsf{NAC}_i}\right)^-\right] = \pi_{ProjColRefs}\left(\sigma_{F_i}\left(r^d_{\mathsf{LHS}} \times \Delta\left(r^d_{\mathsf{NAC}_i}\right)^-\right)\right)$.

For appropriately tracking the above-mentioned modifications of view $r^d_{\mathsf{NAC}_i}$, a delete and an insert operation trigger should be specified by queries $P^-_{\mathsf{NAC}_i}$ and $P^+_{\mathsf{NAC}_i}$ for view r^d_{PRE} on insertion event $\left(\texttt{INSERT}, \Delta\left(r^d_{\mathsf{NAC}_i}\right)^+\right)$ and deletion event $\left(\texttt{DELETE}, \Delta\left(r^d_{\mathsf{NAC}_i}\right)^-\right)$, respectively. Note that there is a negative dependency between modifications in views $r^d_{\mathsf{NAC}_i}$ and r^d_{PRE}, as an insertion into view $r^d_{\mathsf{NAC}_i}$ leads to a deletion in view r^d_{PRE} and vice versa.

8.7 Conclusion

In the current chapter, I elaborated a notification framework based incremental method for graph pattern matching. Additionally, I assessed the performance of the approach by comparing it to a traditional graph transformation tool.

- *Data structures for in-memory incremental graph transformation.* In order to support incremental graph transformation, I proposed data structures for maintaining, efficiently storing, invalidating, and notifying partial matchings, and for accelerating the retrieval of complete matchings (Sec. 8.3).
- *Algorithms for in-memory incremental graph transformation.* By using these data structures, I elaborated algorithms for incremental graph pattern matching, in which complete and partial

matchings of LHS and NAC patterns of a rule are stored explicitly in a snapshot tree in the main memory, and they are updated incrementally when the instance model is modified by also taking into account invalidations due to matchings of negative condition patterns (Sec. 8.4).
- *Quantitative performance analysis of incremental graph transformation.* By using a benchmark example, I examined and compared the run-time performance of the incremental and the traditional graph transformation approaches (Sec. 8.5).
- *Incremental graph transformation in relational databases.* I elaborated a method for incremental graph transformation, which maintains and stores partial matchings of graph transformation rules in relational database tables, which are updated incrementally, when the instance model is changed (Sec. 8.6).

These results are reported in [145, 146, 155, 157].

Relevance

Compared to other graph transformation related incremental techniques, the main distinguishing feature of the presented approach is the novel notification mechanism, which can be characterized by the maintenance of registries for quickly identifying those partial matchings, which are candidates for extension or removal when an object or a link is inserted to or removed from the model.

Users can typically exploit the benefits of incrementality in synchronization tasks. A brief list of such application domains is now given.

- Transformation rules in the Relations language of the Query/Views/Transformations (QVT) [109] standard have multiple domains and these rules can be executed in several directions depending on the target domain being fixed before rule application. If matchings for domains of rules are sought by an incremental graph pattern matcher, the overall model transformation engine can significantly benefit from incrementality.

- A recent study [112] in the field of domain-specific modeling languages suggested the generalization of the mapping between the graphical concrete and the abstract syntax by introducing a declarative framework to give complete freedom to the language engineer in the visualization of models. The implementation of such a declarative framework can also be considered as a synchronization problem, in which incremental graph transformations can provide a suitable technique for improving run-time performance.

- Another recent paper [93] suggests an approach to use a so-called Cognitive Process [94] as a central knowledge-processing entity within artificial cognitive units, which perform co-operative guidance of multiple uninhabited aerial vehicles in assistant systems. This Cognitive Process is implemented by a rule-based approach, for which incremental transformations are applicable to provide an efficient runtime environment.

The Rete network based technique of [22] shows close correspondance to our approach, as levels of snapshot trees can be considered as nodes in the Rete network. Although, it is not a one-to-one mapping as one level of the snapshot tree corresponds to several Rete nodes, two significant consequences can be drawn from this similarity. All techniques (e.g., the handling of common parts of different LHS patterns at the same network node [95]) that have already been invented for Rete-based solutions are also applicable to our approach. The idea of notification arrays can speed-up traditional Rete-based approaches used in a graph transformation context as these arrays help identifying those partial matchings that may participate in the extension of the matching. Thus, it is subject to our future investigations.

Based on the experience collected while carrying out the research reported in the current chapter, a Rete-based incremental pattern matching engine has been developed by a graduate student. This prototype engine now provides an alternative for the traditional, non-incremental pattern matcher module of the VIATRA2 model transformation framework.

The performance analysis of the view-based incremental pattern matching approach of Sec. 8.6 belongs to the future tasks.

Limitations

Certain limitations of the presented algorithms have also been identified.

First of all, the efficiency of the incremental pattern matching engine highly depends on the selection of search plans as even a single insertion (or deletion), which affect matchings located at upper levels of the tree (i.e., near to its root) may trigger computation intensive operations. As a consequence, further investigations on creating good search plans for the incremental pattern matching engine have to be carried out.

Our current solution provides a suboptimal solution, when patterns contain a large number of loop edges. This is related to the fact that our approach currently stores only the matchings of the nodes but not the edges (i.e., edges do not have identifiers), which assumption can be relaxed in the future.

At first glance, it can be strange that NACs are handled independently of the LHS (i.e., all matchings of the NAC are calculated). The goal of our approach is to support the reusability of patterns when the same pattern can be used once in the LHS and once as a NAC, or the same NAC is a negative condition for multiple LHSs (as in VIATRA2 [6]).

CHAPTER

9 Conclusions

As a final conclusion, I compare the results presented in the current thesis with the main objectives (of Sec. 1.4). Additionally, I report on how these results have been used in practical applications. I also outline some future directions of basic research and applications.

9.1 Fulfillment of objectives

Objective 1 After analyzing typical scenarios and the most popular tools from the field of graph transformation, I set up a benchmarking framework by identifying and categorizing the characteristics of the transformation problems themselves and of typical optimization strategies. The proposed framework consists of a model transformation and a simulation benchmark example originating from the software engineering application domain. The framework is used to quantitatively assess the run-time performance of model transformation systems and the acceleration effects of their optimization strategies in practice-oriented environments.

Objective 2 In order to ensure the transformation of large models, I presented a provenly correct method for implementing graph transformation built on top of relational database management systems, which operates on models stored on disks by executing SQL queries and data manipulation commands to perform pattern matching and updating phases, respectively. Additionally, I examined the run-time efficiency of the proposed method on the object-relational mapping benchmark example by using different databases and several parameter and optimization strategy settings. Finally, I extended the method to EJB QL queries to make the RDBMS-based model transformation approach portable and database independent by bridging the gap caused by the different dialects of the SQL standard used in database implementations, and to adhere to the J2EE standard.

Objective 3 In order to improve graph pattern matching heuristics, I introduced model sensitivity by employing statistics collected from concrete typical models of the domain for defining more precise functions for assessing the costs of elementary search plan operations. For the optimization of model-specific search plans, I proposed to customize traditional greedy algorithms.

Moreover, I elaborated an adaptive approach, where the optimal strategy is selected at run-time from precompiled methods by using statistics from the model under transformation.

Additionally, I prepared an EJB3-based prototype of the adaptive graph transformation engine by generating code for pattern matching and cost calculation functionalities of concrete strategies, and by implementing a stateless session bean that selects the optimal strategy at run-time. Finally, I examined and compared the efficiency of Java, EJB3 and EJB QL based pattern matching implementations.

Objective 4 In order to speed-up graph pattern matching for the price of increased memory usage, I proposed data structures and algorithms for incremental graph pattern matching, in which partial matchings of earlier transformation steps are stored explicitly in the main memory, and these matchings are updated incrementally in response to model modification triggers. To avoid exceeding main memory limitations caused by the numerous partial matchings being stored, I additionally elaborated the technique of RDBMS-based incremental pattern matching, in which partial matchings are stored on disk in relational database tables.

Finally, I examined and compared the run-time efficiency of incremental and traditional approaches by using the object-relational benchmark example.

9.2 Utilization of new results

Now I summarize how results of this thesis have been used in practical applications.

9.2.1 Utilization of the benchmarking framework

As reported in [51], the benchmarking framework has been directly used by the developers of GrGen for measuring the run-time performance of their tool. In addition, in [52] they suggested several improvements for the framework itself, for the benchmark implementations, and for chronometry issues. The benchmarking framework is mentioned in several papers [47, 162] and Master's theses [43, 72, 88, 132] as well.

9.2.2 Utilization of RDBMS based graph transformation

The Transformation Execution Environment of the MOLA tool performs RDBMS based graph transformation by generating and executing SQL queries and commands. According to [71], MOLA is similar to our approach presented in Chapter 6 in the sense that both (i) use underlying relational databases, (ii) have a fixed schema, and (iii) perform graph pattern matching by executing SQL queries. In contrast to our approach, which uses a predefined database schema for representing the metamodel, MOLA stores the meta information in tables resulting in more dynamicity and flexibility when the metamodel changes, and slightly more complex queries for pattern matching as type constraints must also be checked by the queries. Negative application conditions are expressed by `NOT EXISTS` subqueries, and not by left joins as in our case. The RDBMS-based graph transformation approach of Chapter 6 has a clear advantage when the metamodel is fixed in advance and if it contains inheritance as type checking can be faster and model consistency can be more easily ensured due to the built-in foreign key constraint support of relational databases.

Contributions that are related to RDBMS based graph transformation have been cited by papers [23, 88, 132].

9.2.3 Utilization of model-sensitive and adaptive pattern matching

Results of Chapter 7 are directly utilized in the development of the VIATRA2 model transformation framework. Adaptive and model-sensitive techniques have already been built into the pattern matching module of Release 3. Unfortunately, the underlying model repository currently lacks the statistical data collection support, which prevented us from testing all the concepts in their full functionality.

The code generation module of FUJABA has recently been improved by putting a stronger emphasis on performance issues as reported in [50]. The original search plan and optimization concepts [164] have been extended by introducing a tree-based representation for search plans, and a sibling permutation heuristic for accelerating pattern matching. Though the current version of the code generation module only uses static cost estimation for search plan operations, the authors mention that the adaptive and model-sensitive approach of Chapter 7 can be easily integrated into their tool as well.

The optimization technique [10, 11] of GrGen is highly similar to the adaptive and model-specific pattern matching approach of Chapter 7 with minor differences in operation cost assignment and search plan cost calculation. The developers have recently confirmed the feasibility of the technique by performing a quantitative analysis reported in [12], which also used the benchmarking framework of Chapter 5. According to [12], the adaptive approach can be an order of magnitude faster than any other known graph transformation systems.

The international acknowledgement of the model-sensitive pattern matching technique is indicated by papers [15, 88, 111].

9.2.4 Utilization of incremental graph pattern matching

Based on the experience on incremental pattern matching techniques, a Rete-based approach has been developed by a graduate student. This prototype engine is now an alternative of the non-incremental pattern matching module of VIATRA2 to be used for domain-specific editors.

Moreover, the technique presented in Chapter 8 have been cited by [79, 93, 115, 138].

9.3 Future directions

An ongoing activity aims at implementing a graph pattern matching module for the VIATRA2 model transformation framework, which is able to handle several advanced pattern composition concepts such as alternate choices and recursion ensuring a scalable and re-usable model transformation engine. A possible research subtask in the development process is the generalization of search plans to support the correct and performance optimal ordering of non-binary constraints.

Further activities aim at integrating traditional and incremental pattern matching engines providing a feature to dynamically adjust time-space trade-off properties of the algorithms based on the actual requirements of the application scenario.

The incremental approach has further potentials to accelerate graph transformation. On one hand, since the definition of subpatterns in Sec. 8.2 corresponds to a linear RETE structure, which is inherently suboptimal, a non-linear layout could improve the performance of consistency restoration. On the other hand, since rules might share subgraph structures in their LHS patterns in a typical application scenario, the RETE nodes that correspond to the common parts can also be merged, thus, reducing the memory consumption of the approach.

Though the adaptive graph transformation technique can theoretically be used in any search plan driven approaches, its widespread usage is set back by the missing statistics support in standard model repositories (like the ones based on EMF). As the introduction of such a support would cause a constant

increase in the complexity of model handling tasks in the repository, this simple step could result in a significant speed-up for model transformations. Additionally, as recent performance experiments [12] showed, there might always be further potential to accelerate pattern matching by developing new heuristics for model-sensitive search plans.

Finally, since graph pattern matching has several independently executable subtasks, the development of a parallel and distributed graph transformation engine could be a new direction of research in the future.

APPENDIX

A Proofs of Theorems

Theorem 1 *The initial instance model M and its database representation $\mathfrak{M}$ are consistent (see Def. 51). Formally, $M \cong \mathfrak{M}$.*

PROOF In order to prove the consistency of M and $\mathfrak{M}$, we have to check whether statements in Definition 51 hold in both directions for all classes and associations.

Nodes. $\Longrightarrow$ First we check the property that should be hold for the classes. Let us select an arbitrary class $C \in V_{MM}$.

According to the left part of Def. 51, $\exists c \in V_M$ such that $C \overset{*}{\leftarrow} t(c)$. Since topological order (Def. 49) enumerates all the ancestors of $t(c)$, C will surely appear in the topological order of $t(c)$. But Alg. 6.1 iterates over all objects (lines 1–6), then over all classes appearing in the topological order (lines 3–5), line 4 is also executed for the object c, class C pair, which means that the identifier c^d generated for c in line 2 should be contained by table C^d in column id after the termination of Alg. 6.1. The same statement is valid for any arbitrary class of the metamodel.

Many-to-one edges. $\Longrightarrow$ Now we have a many-to-one link $a \overset{e}{\rightarrow}_1 b \in E_M$. When Alg. 6.1 reaches line 7, the source object a of this link has already a database representation, which means that there exists a row $\vec{a}$ with $\vec{a}[id] = a^d$ in all tables that correspond to ancestors of class $t(a)$. As $src(t(e)) \overset{*}{\leftarrow} t(a)$ holds according to the type conformance requirements of Def. 7 for source objects, there exists a row $\vec{a}$ with $\vec{a}[id] = a^d$ in table $src(t(e))^d$. But the update operation in line 8 of Alg. 6.1 is executed for our selected many-to-one link, which sets $\vec{a}[t(e)^d]$ to b^d, thus we have found an appropriate row $\vec{a}$ required by Def. 51.

Many-to-many edges. $\Longrightarrow$ It can be assumed that we have a many-to-many link $a \overset{e}{\rightarrow}_* b \in E_M$. Since lines 10–12 are executed for all many-to-many links of the instance model, it should also be executed for $a \overset{e}{\rightarrow}_* b$ as well, which includes the insertion of tuple (a^d, b^d) to table $t(e)^d$ in line 11. But we are ready now, since (a^d, b^d) got into the table $t(e)^d$ as it is required in the right side of Def. 51.

Nodes. $\Longleftarrow$ Let us select an arbitrary class $C \in V_{MM}$ again. By using the statement of consistency definition (Def. 51) for a class C, it may be assumed that $\exists \vec{c} \in C^d$ such that $\vec{c}[id] = c^d$, thus there is a row $\vec{c}$ in table C^d that contains the value c^d in column id. Since table C^d was empty in the beginning, the only possibility for c^d to appear in the table is that it should be inserted during the execution of lines 1–6 of Alg. 6.1. But this could only happen, if object c and class C have been enumerated in line 1 and in line 3, respectively. Since class C has to be in the topological order of $t(c)$, this means

that $C \overset{*}{\leftarrow} t(c)$. But in this case we have found an object c for which $C \overset{*}{\leftarrow} t(c)$ holds, so it fulfils the requirements appearing in the left part of Def. 51. Since in the beginning an arbitrary class was selected, our proof is valid for all other classes as well.

Many-to-one edges. $\Longleftarrow$ It can be assumed that table $\mathcal{T}$, which corresponds to a class in the metamodel, has a row $\vec{a}$ for which $\vec{a}[id] = a^d$ and $\vec{a}[t(e)^d] = b^d$ hold. Since all tables were initially empty and only line 8 of Alg. 6.1 is able to modify such table $\mathcal{T}$ in columns other than id, this part of the algorithm has to be executed. But this can only happen, if there exists a many-to-one link $a \xrightarrow{e}_1 b$ in model M.

Many-to-many edges. $\Longleftarrow$ We know that there exists a row $\vec{e} = (a^d, b^d)$ in a table $t(e)^d$. Since tables were empty initially, $\vec{e}$ had to be inserted during one execution of lines 10–12 of Alg. 6.1, which means that there should exist a many-to-many link $a \xrightarrow{e}_* b$ in the original instance model M for which the corresponding `INSERT` operation could be executed in line 11. □

Theorem 2 *Let d be a bidirectional mapping between $\mathfrak{S}_{GT}$ and $\mathfrak{S}_{DB}$. If model M is consistent with the database representation $\mathfrak{M}$, then a pattern r_G (*without *negative application condition) in $\mathfrak{S}_{GT}$ is consistent with view r_G^d in $\mathfrak{S}_{DB}$. Formally, $M \cong \mathfrak{M} \Longrightarrow r_G \cong r_G^d$.*

PROOF ($\Longrightarrow$) When proving in this direction, we may assume that we have a matching m for rule graph r_G in model M, and we want to prove that there exists a corresponding row in view r_G^d.

Since $M \cong \mathfrak{M}$ we know that the instance model has a correct representation in the database. During the proof we first examine what the contents of database tables are, and then we apply operations defined in the query for r_G^d step-by-step, and our aim is to prove that the result (namely the r_G^d view) will contain a row $\vec{r}$ with object identifiers defined by matching m.

Consequences of $M \cong \mathfrak{M}$. Having a matching m means that for all nodes and edges of the G graph have a type conform image in the model M.

Let us use the consistency definition (Def. 51) in left to right direction for any object $m(x) \in V_M$ that participates in the matching m. We get that a corresponding row $\vec{m}_x$ with $\vec{m}_x[id] = m(x)^d$ should be contained not only by table assigned to its own direct type $t(m(x))^d$ but also by all its ancestor tables, and as such $\vec{m}_x \in t(x)^d$ as well. By applying the consistency definition for many-to-one link $a \xrightarrow{e}_1 b$ assigned to an edge $u \xrightarrow{z}_1 v$ of rule graph G by matching m, we get that table $src(t(e))^d$ has a row $\vec{m}_z$ for which $\vec{m}_z[id] = a^d$ and $\vec{m}_z[t(e)^d] = b^d$ hold. Since $t(e) = t(z)$, $\vec{m}_z$ appears in $src(t(z))^d$ as well. By using the consistency definition for many-to-many link $a \xrightarrow{e}_* b$ assigned to an edge $u \xrightarrow{z}_* v$ of rule graph G by matching m, we get that table $t(e)^d$ has a row $\vec{m}_z = (a^d, b^d)$. It is worth to emphasize that at this point we already know the contents of all database tables that are used in the query of r_G^d.

Construction of the joined table. Now, if we enumerate nodes and edges of G in their natural order (and also take care of nodes being ahead of edges in the enumeration), and we select exactly the same rows from the tables that were mentioned above, then a row $\vec{s} = \left(\vec{m}_{x_1}, \ldots, \vec{m}_{x_{n_V}}, \vec{m}_{z_1}, \ldots, \vec{m}_{z_{n_E}}\right)$ will appear in the joined table $\mathcal{T} = t(x_1)^d \times \cdots \times t(x_{n_V})^d \times t(z_1)^d \times \cdots \times t(z_{n_E})^d$. In the following, it is examined why row $\vec{s}$ is not filtered out by injectivity and edge constraints of the selection operation.

Checking injectivity constraints. Let us suppose by contradiction that $\vec{s}$ has been filtered out because of violating an injectivity constraint in the query (e.g. $x_j^{cs}.id \neq x_k^{cs}.id$ for some different $x_j, x_k \in V_G$ where $t(x_j) \overset{*}{\leftarrow} t(x_k)$ holds). Violating the constraint means that values should be equal in columns $x_j^{cs}.id$ and $x_k^{cs}.id$ for all rows the joined table contains, and as such this equation must also hold for the corresponding elements of $\vec{s}$. By taking care of construction rules of $\vec{s}$ it yields to

$m(x_j)^d = \vec{m}_{x_j}[id] = \vec{m}_{x_k}[id] = m(x_k)^d$. Since d is bijective, the equation could hold only if, the origins in model M were the same ($m(x_j) = m(x_k)$). But in this case we have different rule graph nodes that have been mapped to the same object of the model by m, which is an immediate violation of injective mapping requirements for m. As a consequence, we may state that if m takes care of injective mapping, then the injectivity filtering condition will also take care of this requirement for the database representation.

Checking edge constraints. Let us select an arbitrary many-to-one edge $u \xrightarrow{z}_1 v \in E_G$ and let us further suppose that it is mapped to link $a \xrightarrow{e}_1 b$ by matching m. As a consequence of the query construction algorithm, we know that $\vec{s}[z^{cs}.id] = \vec{m}_z[id] = a^d$, and similarly, $\vec{s}[z^{cs}.t(z)^d] = \vec{m}_z[t(z)^d] = b^d$. Since u and v are rule graph nodes in G, there should exist columns $\vec{s}[u^{cs}.id]$ and $\vec{s}[v^{cs}.id]$ originating from $\vec{m}_u[id]$ and $\vec{m}_v[id]$ with values a^d and b^d, respectively. Summarizing our previous statements result in $\vec{s}[u^{cs}.id] = a^d = \vec{s}[z^{cs}.id]$ and $\vec{s}[v^{cs}.id] = b^d = \vec{s}[z^{cs}.t(z)^d]$. Recall the edge constraint that has been defined for edge $u \xrightarrow{z}_1 v$. Note that this specific edge constraint prescribes the equation of exactly the same columns, whose equation has just been proved for $\vec{s}$.

Let us select an arbitrary many-to-many edge $u \xrightarrow{z}_* v \in E_G$ and let us further suppose that it has been mapped to $a \xrightarrow{e}_* b$ by matching m. By using a similar reasoning, we get equalities $\vec{s}[u^{cs}.id] = a^d = \vec{s}[z^{cs}.src]$ and $\vec{s}[v^{cs}.id] = b^d = \vec{s}[z^{cs}.trg]$, which means that $\vec{s}$ fulfils the edge constraints defined for edge $u \xrightarrow{z} v$.

Since $\vec{s}$ satisfies all the injectivity and edge constraints we may state that $\vec{s} \in \sigma_{Inj \wedge Edge}(\mathcal{T})$.

Performing projection. By using the definition of projection to columns being defined in Sec. 6.4.2, we get $\vec{r} = \left(m(x_1)^d, \ldots, m(x_{n_V})^d\right) \in r_G^d$, which means that we have found a row in r_G^d that contains all the identifiers of nodes that have been selected by the specific matching. □

PROOF ($\Longleftarrow$) When proving in this direction, we may assume that table r_G^d having n_V columns contains a row $\vec{r}$, for which $\forall x \in V_G : \vec{r}[x^d] = c^d$. Now our goal is to define an appropriate matching m for rule r_G in model M.

In this case the idea of the proof goes rather in a backward direction. We already know that the joined table $\mathcal{S}$ contains a row $\vec{s}$ from which $\vec{r}$ could originate during its calculation, but since the joined table has more columns than the result table, some values in row $\vec{s}$ are unknown initially. By using edge constraints, we are able to calculate some further values, resulting in a row $\vec{s}$ that has more values filled in than $\vec{r}$. Then we define the matching m based on the values in row $\vec{s}$, and finally we prove that this matching must also satisfy injectivity constraints together with its original database representation.

Following the projection and selection operations in backward direction. Now we have a row $\vec{r}$ in r_G^d. If an operation (such as projection and selection) cannot increase the number of rows, then it is sure that if we have a row in the result table, then this row should have an origin in the table, on which operations were performed. Formally, it is obvious (by using the definitions of projection and selection) that $\exists \vec{s} \in \sigma_{Inj \wedge Edge}(\mathcal{S}) \subseteq \mathcal{S} = \mathcal{T}_1 \times \cdots \times \mathcal{T}_{n_V + n_E}$, where $\mathcal{T}_i$ is the table that corresponds to the ith graph object (node or edge) of the pattern G as defined by the query construction algorithm. By investigating the columns to which projection was applied, we can calculate what the values of row $\vec{s}$ should have been before the projection was performed. More precisely, $\forall x \in V_G : c^d = \vec{r}[x^d] = \vec{s}[x^{cs}.id]$.

Matching definition for rule graph nodes. Let us examine an arbitrary node x of pattern G. According to the definition of $\mathcal{S}$, the column set x^{cs} that corresponds to x should originate from table $t(x)^d$ that was assigned to class $t(x)$. As a consequence, there should exist a row $\vec{t}_x$ in table $t(x)^d$ such that $\vec{s}[x^{cs}.id] = \vec{t}_x[id] = c^d$. Since our tables contain unique identifiers of objects in columns id, there should exist a single object c whose identifier is c^d. Now the consistency definition (Def. 51) can be used in right to left direction, which means that the direct type $t(c)$ of object c is a descendant of $t(x)$,

so it is allowed to map node x to object c by matching m. So we can define the matching m for rule graph node x as $m(x) := c$.

Matching definition for many-to-one rule graph edges. Let us select an arbitrary many-to-one edge $u \xrightarrow{z}_1 v$ from pattern G. Recall how edge constraints look like for this specific edge. These constraints are $z^{cs}.id = u^{cs}.id$, and $z^{cs}.t(z)^d = v^{cs}.id$. Note that since u and v are nodes in pattern G, $\vec{s}[u^{cs}.id]$ and $\vec{s}[v^{cs}.id]$ have some values a^d and b^d being identifiers of objects a and b, respectively, as we determined earlier. Furthermore, we know that $t(u) \overset{*}{\leftarrow} t(a)$ and $t(v) \overset{*}{\leftarrow} t(b)$. Edge constraints must hold for all rows of $\mathbb{S}$ and as such $\vec{s}$ should also satisfy them, resulting in $\vec{s}[z^{cs}.id] = \vec{s}[u^{cs}.id] = a^d$ and $\vec{s}[z^{cs}.t(z)^d] = \vec{s}[v^{cs}.id] = b^d$. We know that the column set z^{cs} of $\mathbb{S}$ should originate from the table $src(t(z))^d$ that was assigned to class $src(t(z))$. Since $\vec{s}$ is in the joined table $\mathbb{S}$, $src(t(z))^d$ should have a row $\vec{t}_z$ such that $\vec{t}_z[id] = \vec{s}[z^{cs}.id] = a^d$ and $\vec{t}_z[t(z)^d] = \vec{s}[z^{cs}.t(z)^d] = b^d$. The consistency definition (Def. 51) for many-to-one links in right to left direction states that $\exists a \xrightarrow{e}_1 b \in E_M$ such that $t(z) = t(e)$. But this edge is an appropriate candidate to which pattern edge $u \xrightarrow{z}_1 v$ can be mapped by matching m.

Matching definition for many-to-many rule graph edges. Let us select an arbitrary many-to-many edge $u \xrightarrow{z}_* v$ from pattern G. Edge constraints for this specific edge are $z^{cs}.src = u^{cs}.id$ and $z^{cs}.trg = v^{cs}.id$. Since u and v are nodes of pattern G, $\vec{s}[u^{cs}.id]$ and $\vec{s}[v^{cs}.id]$ have some values a^d and b^d that are identifiers of objects a and b, respectively. Moreover, we know that $t(u) \overset{*}{\leftarrow} t(a)$ and $t(v) \overset{*}{\leftarrow} t(b)$. Edge constraints must be satisfied by row $\vec{s}$, which means that $\vec{s}[z^{cs}.src] = \vec{s}[u^{cs}.id] = a^d$ and $\vec{s}[z^{cs}.trg] = \vec{s}[v^{cs}.id] = b^d$ should hold. We know that column set z^{cs} of $\mathbb{S}$ derives from table $t(z)^d$, which has been created for association $t(z)$. Since $\vec{s}$ is in table $\mathbb{S}$, there should exist a row $\vec{t}_z$ in table $t(z)^d$ such that $\vec{t}_z[src] = \vec{s}[z^{cs}.src] = a^d$ and $\vec{t}_z[trg] = \vec{s}[z^{cs}.trg] = b^d$. The consistency definition (Def. 51) for many-to-many links in right to left direction states that there exists a link $a \xrightarrow{e}_* b \in E_M$ such that $t(z) = t(e)$. Now we may define matching m for edge $u \xrightarrow{z}_* v$ as $m(u \xrightarrow{z}_* v) := a \xrightarrow{e}_* b$.

Injectivity constraint check. Finally, we check that the matching m we have just defined cannot map different nodes (edges) to the same object (link).

Let us suppose by contradiction, that there are two different nodes x_j, x_k in G such that $t(x_j) \overset{*}{\leftarrow} t(x_k)$ and m maps them to the same object c. Formally, $m(x_j) = m(x_k) = c$. Since d is bijective, these objects have the same identifier in the database, formally $m(x_j)^d = m(x_k)^d = c^d$. We have some further knowledge about this identifier, namely $\vec{s}[x_j^{cs}.id] = c^d = \vec{s}[x_k^{cs}.id]$. Recall that injectivity constraints prescribed inequality for exactly the same columns, namely $x_j^{cs}.id \neq x_k^{cs}.id$. Injectivity constraints should be satisfied by row $\vec{s}$ in order to be the origin of row $\vec{r}$, which is a contradiction, since we found equality of elements in the mentioned columns in case of row $\vec{s}$.

Different pattern edges cannot be mapped to the same link, as in such a situation the pattern could not be a well-formed instance of the metamodel, since it would violate the non-existence of parallel edges. □

Corollary 1 *If we calculate the left outer join of tables $\mathbb{R}^{(m)}$ and $\mathbb{S}^{(n)}$, then for each row $\vec{r}$ of $\mathbb{R}$ there exists a row $\vec{t}$ in the joined table that contains row $\vec{r}$ in its first m columns. Formally, if $\mathbb{T} = \mathbb{R} \overset{F}{\ltimes} \mathbb{S}$ then $\forall \vec{r} \in \mathbb{R}, \exists \vec{t} \in \mathbb{T}$ such that $\vec{t}[i] = \vec{r}[i]$ for all the columns of $\vec{r}$.*

In the following, notation $\mathbb{S}_i$ will be used for $r^d_{\mathsf{LHS}} \overset{F_1}{\ltimes} r^d_{\mathsf{NAC}_1} \overset{F_2}{\ltimes} \dots \overset{F_i}{\ltimes} r^d_{\mathsf{NAC}_i}$. With this notation $\mathbb{S}_k$ corresponds to the table that has to be calculated for the view r^d_{PRE}.

Theorem 3 *Let us suppose that there exists a bijective mapping from $\mathfrak{S}_{GT}$ to $\mathfrak{S}_{DB}$. If model M is consistent with the database representation $\mathfrak{M}$, then a pattern r_{PRE} in $\mathfrak{S}_{GT}$ that* has *negative application conditions is consistent with view r^d_{PRE} in $\mathfrak{S}_{DB}$. Formally, $M \cong \mathfrak{M} \Longrightarrow r_{PRE} \cong r^d_{PRE}$.*

PROOF ($\Longrightarrow$) The basic idea is to prove that $\mathcal{S}_k$ should contain a row $\vec{s}$ that has defined values only in columns that originate from view r^d_{LHS}, and all other values are undefined. This is done in an iterative process starting from $\mathcal{S}_0$, which corresponds to view r^d_{LHS}. In each step in order to generate $\mathcal{S}_i$, $r^d_{\mathsf{NAC}_i}$ is attached to $\mathcal{S}_{i-1}$ by a left outer join operation using the formulae F_i for join condition. Finally, we show that the projection and selection performed in the last phases of r^d_{PRE} calculation does not filter out row $\vec{s}$ from the set of results, yielding to an appropriate row $\vec{r}$ in view r^d_{PRE}.

Since m is a matching for pattern r_{PRE}, it is also a matching for r_{LHS}. By using Theorem 2, this means that $\exists \vec{t_0} \in r^d_{\mathsf{LHS}} = \mathcal{S}_0$.

Lemma. Let us suppose by induction that we have already calculated $\vec{t}_{i-1} \in \mathcal{S}_{i-1}$ and $\vec{t}_{i-1} = (\vec{t_0}[x^d_1], \ldots, \vec{t_0}[x^d_{n_V}], \varepsilon, \ldots, \varepsilon)$. In other words the first n_V columns of $\vec{t}_{i-1}$ contains the same values as $\vec{t_0}$, while all the remaining values are undefined. We want to prove that $\vec{t_i}$ has similar structure and that $\vec{t_i}$ can also be found in table $\mathcal{S}_i$.

Proof of the lemma. Let us calculate $\mathcal{S}_i$. By using Corollary 1, it can be stated that columns of $\vec{t_i}$ that originate from $\mathcal{S}_{i-1}$ have the same values as $\vec{t}_{i-1}$ independently of the fact whether the join condition F_i holds or not. The only thing to be checked is whether the last n_{V_i} columns of $\vec{t_i}$ (originating from $r^d_{\mathsf{NAC}_i}$) are filled with undefined values.

Let us suppose by contradiction that there exists $\vec{r_i}$ in view $r^d_{\mathsf{NAC}_i}$ that can be attached to $\vec{t}_{i-1}$ by left outer join in such way that F_i holds. By using Theorem 2 there should exist a matching m' for the graph objects of r_{NAC_i}.

If x is an arbitrary shared node of NAC_i with an origin x_l in the LHS (thus $x_l \in \underline{V_{\mathsf{LHS}}} \cap V_{\mathsf{NAC}_i}$, $x \in \underline{V_{\mathsf{NAC}_i}} \cap V_{\mathsf{LHS}}$, and $p_{\mathsf{NAC}_i}(x_l) = x$), then because of the construction algorithm of views r^d_{LHS} and $r^d_{\mathsf{NAC}_i}$, they have a column that represents node x_l and its shared node image x, respectively. But we assumed that F_i is satisfied, which means that $r^{cs}_{\mathsf{LHS}}.x^d_l = r^{cs}_{\mathsf{NAC}_i}.x^d$ should hold for all the rows, and as such for $\vec{t_i}$ as well. By summarizing our knowledge about $\vec{t_i}$ we get

$$\vec{t_0}[x^d_l] = \vec{t}_{i-1}[r^{cs}_{\mathsf{LHS}}.x^d_l] = \vec{t_i}[r^{cs}_{\mathsf{LHS}}.x^d_l] = \vec{t_i}[r^{cs}_{\mathsf{NAC}_i}.x^d] = \vec{r_i}[x^d].$$

$\vec{t_0}[x^d_l]$ and $\vec{r_i}[x^d]$ define the identifiers of objects to which x was mapped by m and m', respectively. Thus, $m(x_l)^d = \vec{t_0}[x^d_l] = \vec{r_i}[x^d] = m'(x)^d$. Since d is bijective, $m(x_l) = m'(x)$, which means that each shared node of NAC_i had to be mapped onto the same object, which was assigned to their origin in LHS.

At this point we know that all the shared nodes of NAC_i and their origins in LHS are mapped to the same objects by matchings m' and m, respectively. If the definition of matching for rule r_{PRE} is recalled from Sec. 6.4.3, then it can be seen that m cannot be a matching, since m and m' together violate the second part of the definition, which prohibits the existence of a matching for NAC_i. So our initial assumption to have a row $\vec{r_i}$ that satisfies F_i together with $\vec{t}_{i-1}$ failed. But if there are no such row $\vec{r_i}$ for which F_i could hold, then only the second part of the left join definition could have been used when calculating $\vec{t_i}$, which means that the columns of $\vec{t_i}$ originating from $r^d_{\mathsf{NAC}_i}$ must be padded with undefined values. At this point we may conclude that we have found a row $\vec{t_i}$ in view $\mathcal{S}_i$ that has the prescribed structure. □

Consequence of the lemma. By using our lemma k times, we get that there is a row $\vec{t_k} \in \mathcal{S}_k$, which contains defined values in columns originating from view r^d_{LHS} and all the other values are undefined.

The effect of selection. Since null conditions $Null$ of the selection operation pose restrictions only on columns originating from negative application condition views $r^d_{\mathsf{NAC}_i}$, $\vec{t_k}$ surely satisfies all of them, since it contains undefined values in all such columns.

The effect of projection. The last operation is the projection, which selects the first n_V columns of $\vec{t_k}$ resulting in a row $\vec{r} \in r^d_{\mathsf{PRE}}$. Note that the first n_V columns of $\vec{t_k}$ are the ones that contain identifiers

originating from r^d_{LHS}, and they are never undefined. It can be now concluded that a row $\vec{r}$ is found in the view that represents r_{PRE}. □

PROOF ($\Longleftarrow$) We know that there exists a row $\vec{r}$ in view r^d_{PRE} and an appropriate matching m for rule r_{PRE} is to be found.

Proof by contradiction I. Let us suppose by contradiction that we have $\vec{r} \in r^d_{\mathsf{PRE}}$, but no matching m exists for the LHS rule graph (r_{LHS}).

If no matchings exist for r_{LHS}, then Theorem 2 yields to an empty r^d_{LHS} view. But note that this view appears at the leftmost position of left join operations in the definition of r^d_{PRE}, which means that r^d_{PRE} should also be empty. But this contradicts to our initial assumption, since $\vec{r} \in r^d_{\mathsf{PRE}}$.

Proof by contradiction II. Let us suppose by contradiction that we have $\vec{r} \in r^d_{\mathsf{PRE}}$, and a matching m for r_{LHS}, and there is also a matching m' for a rule graph r_{NAC_i} such that each node and edge are mapped to the same object and link, respectively, by both m and m'.

By using Theorem 2 for matchings m and m', we get that $\vec{s_0} \in \mathcal{S}_0 = r^d_{\mathsf{LHS}}$ and $\vec{r_i} \in r^d_{\mathsf{NAC}_i}$. Let us suppose that row $\vec{s_k}$ of view $\mathcal{S}_k$ was calculated by using $\vec{s_0}$ and $\vec{r_i}$. For the sake of simplicity, let us focus only on columns of $\vec{s_k}$ that originate from $r^d_{\mathsf{NAC}_i}$. Our statement is that this portion of $\vec{s_k}$ agrees with $\vec{r_i}$.

The portion of $\vec{s_k}$ originating from $r^d_{\mathsf{NAC}_i}$ is introduced when $\mathcal{S}_i$ is calculated, and afterwards it is left unchanged by left outer join operations. But when the ith left outer join is executed its join condition F_i holds, and in this case inner join has to be executed resulting in our statement mentioned above. The only thing to be checked is why F_i is satisfied. Note that F_i is defined on the shared nodes of r_{NAC_i}, and their corresponding origins in r_{LHS}. Each shared node x and its origin x_l is mapped to the same object c by matchings m' and m, respectively, so $\vec{s_{i-1}}[r^{cs}_{\mathsf{LHS}}.x^d_l] = c^d = \vec{r_i}[x^d]$, which means that we found correspondence in all columns of $\vec{s_{i-1}}$ and $\vec{r_i}$ for which correspondence was prescribed by F_i.

Note that null conditions require the image of shared nodes of r_{NAC_i} to be undefined in columns of $\vec{s_k}$ that originate from r_{NAC_i}, which is immediately violated, since they got their values just in the previous paragraph. So $\vec{s_k}$ violates null conditions of the selection operation, and as a consequence it should be filtered out inhibiting $\vec{s_k}$ to be the origin of $\vec{r}$. It means that under the supposed circumstances no origin of $\vec{r}$ exists in view $\sigma_{Null}(\mathcal{S}_k)$, which is a contradiction.

Final consequence. At this point we know that there should exist a matching m for r_{LHS}, but no matching m' for any r_{NAC_i}. Recalling the definition of matching of r_{PRE}, we get that the above-mentioned situation is the one that fulfills all the requirements, so matching m is also good for r_{PRE}. □

Theorem 4 *Let us suppose that there exists a bijective mapping d from $\mathfrak{S}_{GT}$ to $\mathfrak{S}_{DB}$. If (i) model M is consistent with the database representation $\mathfrak{M}$, (ii) we have a matching m_r for rule r, together with a corresponding row $\vec{m}^d$ in view r^d, and m is consistent with $\vec{m}^d$, (iii) rule r is applied on matching m_r resulting in M', and (iv) Algorithms 6.2–6.5 are executed in the database for $\vec{m}^d \in r^d$ resulting in a database representation $\mathfrak{M}'$, then $M' \cong \mathfrak{M}'$.*

Formally, if

(i) $M \cong \mathfrak{M}$,
(ii) $(m_r|r) \cong (\vec{m}^d|r^d)$ *for a pair* $(m_r, \vec{m}^d)$,
(iii) $M \overset{r,m_r}{\Longrightarrow} M'$,
(iv) $\mathfrak{M} \overset{Alg.\ 6.2-6.5}{\Longrightarrow} \mathfrak{M}'$,

then $M' \cong \mathfrak{M}'$.

PROOF The model manipulation phase of a rule application can be divided into a deletion and an insertion step. Our first goal is to prove that context model M_c is consistent with database representation $\mathfrak{M}_c$ resulted by the execution of Alg. 6.2 and 6.3. Then the consistency of derived model M' and database representation $\mathfrak{M}'$ is proven based on the consistency of M_c and $\mathfrak{M}_c$. Since skeletons of the proofs are exactly the same in these steps, we only present the technique for the more difficult (i.e., the deletion) step.

The proof of the deletion step is bidirectional and it has 7 cases in each direction, which use exactly the same technique and which have to be checked one by one. The 7 cases correspond to the deletion of (i) objects; (ii) many-to-one and (iii) many-to-many dangling links leaving an object to be deleted; (iv) many-to-one and (v) many-to-many dangling links leading into an object to be deleted; and (vi) many-to-one and (vii) many-to-many links selected by matching m for an edge $z \in E_{\mathsf{LHS}} \setminus E_{\mathsf{RHS}}$. We may identify a well-defined part of Alg. 6.2 and 6.3 for each case where the specific case is handled by these algorithms in the database. Table A.1 presents the cases and their corresponding handling routines.

Case	Object/link	Reason of selection	DB operation
(i)	object	selected by m	line 12 of Alg. 6.3
(ii)	many-to-one link	dangling/outgoing	line 12 of Alg. 6.3
(iii)	many-to-many link	dangling/outgoing	line 4 of Alg. 6.3
(iv)	many-to-one link	dangling/incoming	line 10 of Alg. 6.3
(v)	many-to-many link	dangling/incoming	line 7 of Alg. 6.3
(vi)	many-to-one link	selected by m	line 2 of Alg. 6.2
(vii)	many-to-many link	selected by m	line 5 of Alg. 6.2

Table A.1: Different cases and corresponding lines of Alg. 6.2 and 6.3 participating in the proof

In order to avoid tedious and lengthy proofs of the same style, we only sketch the skeleton of the proof technique and we present a complete proof for only one case (i.e., for the deletion of nodes) in both directions. The proofs for the other cases can be derived from the presented complete proof by replacing object and lines of Algorithms 6.2 and 6.3 by a corresponding kind of link and lines of the same algorithms, respectively, as defined in Table A.1 for the given case.

PROOF ($\Longrightarrow$) The skeleton of the proof is as follows. We select an object (a link) from context model M_c. Since only deletions are performed on model M, M should also contain the same object (link). Then the consistency of M and $\mathfrak{M}$ is used in left to right direction to ensure that the object (link) is represented in the database (i.e., in $\mathfrak{M}$). Finally, it is examined why the database representation of the object (link) cannot be deleted from $\mathfrak{M}$ during the execution of Alg. 6.2 and 6.3.

Nodes. Let us select an object c from context model M_c and an arbitrary class $C \in V_{MM}$ such that $C \overset{*}{\leftarrow} t(c)$. Object c has to appear in model M as only deletions have been performed on model M in the deletion phase. By using the consistency of model M and database representation $\mathfrak{M}$ (Def. 51) for objects in left to right direction, we get that $\exists \vec{c} \in C^d$ such that $\vec{c}[id] = c^d$.

The only position where either Alg. 6.2 or Alg. 6.3 can delete row $\vec{c}$ from C^d is line 12 of Alg. 6.3. (All other database operations either delete rows from tables assigned to many-to-many associations, or updates tables assigned to classes in columns not equal to id.) Line 12 of Alg. 6.3 would delete row $\vec{c}$, if $\exists x \in V_{\mathsf{LHS}} \setminus V_{\mathsf{RHS}}$ such that $m(x) = c$, but the existence of such node x would yield to the deletion of object c from model M, which is impossible as context model M_c still contains c. The result of this

reasoning is that $\vec{c}$ could not be deleted by Alg. 6.2 and 6.3, which means that $\vec{c} \in C^d$ also in database $\mathfrak{M}_c$. $\square$

PROOF ($\Longleftarrow$) Now the proof proceeds in the other direction. We have a row in a table of $\mathfrak{M}_c$, which was assigned to a class (many-to-many association). Since Alg. 6.2 and 6.3 can delete rows or set undefined values to columns with name not equal to id, src, trg, it is sure that a row with the same value in column id (in columns src and trg) can be found in the same table of $\mathfrak{M}$. In this case, we may apply the consistency of $\mathfrak{M}$ and M for objects or many-to-one links (for many-to-many links) in right to left direction, resulting in a corresponding object or many-to-one link (many-to-many link) in model M. Finally, it is investigated why this object or many-to-one link (many-to-many link) is not deleted in the deletion phase of GT rule application.

Nodes. We have a row $\vec{c'} \in C^{d'}$ with $\vec{c'}[id] = c^d$ where $C^{d'}$ represents a table that was assigned to a class $C \in V_{MM}$ and that has a content according to the database representation $\mathfrak{M}_c$. Since only row deletions and updates in columns with name not equal to id could be performed on table C^d during the execution of Alg. 6.2 and 6.3, it is sure that $\exists \vec{c} \in C^d$ such that $\vec{c}[id] = \vec{c'}[id] = c^d$. By using the consistency of model M and database representation $\mathfrak{M}$ (Def. 51) for objects in right to left direction, we get that $\exists c \in V_M$ such that $C \overset{*}{\leftarrow} t(c)$.

Let us suppose by contradiction that there is a node $x \in V_{\mathsf{LHS}} \setminus V_{\mathsf{RHS}}$ such that $m(x) = c$ and $t(x) \overset{*}{\leftarrow} t(c)$. Since $C \overset{*}{\leftarrow} t(m(x)) = t(c)$, class C should have been enumerated in the inverse topological order of $t(m(x))$, and as a consequence, line 12 of Alg. 6.3 should have been executed on table C^d with condition $id = c^d$, which means that $\vec{c}$ should have been removed, as $\vec{c}[id] = c^d$. This is a contradiction, since $\vec{c}$ remained in table C^d in database content $\mathfrak{M}_c$.

So $\nexists x \in V_{\mathsf{LHS}} \setminus V_{\mathsf{RHS}}$ that is mapped to c by m. But in this case c is not removed from model M, thus, c remains in context model M_c. $\square$

APPENDIX

B Additional Algorithms

Algorithm B.1 The snapshot deletion algorithm `delete`(s_{G_k})

PROCEDURE `delete`(s_{G_k})

1: **if** $G_0 \subset G_k$ **then**
2: $\quad \mathbb{S}_G^{\mathcal{T}\prime} := \mathbb{S}_G^{\mathcal{T}} \setminus \{ s_{G_k} \}$
3: $\quad$ `removeDeleteEntries`(s_{G_k})
4: $\quad$ **for all** $(s, s_{G_k}) \in \mathcal{I}$ **do**
5: $\quad\quad \mathcal{I}' := \mathcal{I} \setminus \{ (s, s_{G_k}) \}$
6: $\quad$ **end for**
7: $\quad$ `invalidate`(s_{G_k})
8: **end if**

Algorithm B.2 The snapshot invalidation algorithm `invalidate`(s_{G_k})

PROCEDURE `invalidate`(s_{G_k})

if $G_k = G$ **then**
  {If $s^m_{G_k}$ is a *(complete) matching* for pattern G}
  if $G = \mathsf{LHS}$ **then**
    {If $s^m_{G_k}$ is a (complete) matching for an LHS pattern}
    $R'_G := R_G \setminus \{ s_{G_k} \}$ {Remove the snapshot from the results}
  else
    {If $s^m_{G_k}$ is a (complete) matching for a NAC pattern}
    for all $\left\{ s \in \mathbb{S}^{\mathcal{T}}_{\mathsf{LHS}} \mid (s_{G_k}, s) \in \mathcal{I} \right\}$ **do**
      if $\forall s_{\mathsf{NAC}} \in \bigcup_i \mathbb{S}^{\mathcal{T}}_{\mathsf{NAC}_i} : (s_{\mathsf{NAC}}, s) \notin \mathcal{I} \vee s_{\mathsf{NAC}} = s_{G_k}$ **then**
        `validate`(s) {If snapshot s was invalidated only by s_{G_k}, then validate s.}
      end if
      $\mathcal{I}' := \mathcal{I} \setminus \{ (s_{G_k}, s) \}$
    end for
  end if
else
  {If $s^m_{G_k}$ is a *partial* matching for pattern G}
  `removeInsertEntries`(s_{G_k}) {Remove insert entries}
  `propagateDelete`(s_{G_k})
end if

Algorithm B.3 The `copyMatchings`(s_{G_k}, c) method

PROCEDURE `copyMatchings`(s_{G_k}, c)
Require: $G_k \subset G$
1: $\mathbb{S}_G^{\mathcal{T}'} := \mathbb{S}_G^{\mathcal{T}} \cup \left\{ s_{G_{k+1}} \right\}$
2: $p(s_{G_{k+1}}) := s_{G_k}$
3: **for all** $v \in V_{G_k}$ **do**
4: $s^m_{G_{k+1}}(v) := s^m_{G_k}(v)$ {Mappings of pattern nodes contained by the kth subpattern are copied to the new matching $s^m_{G_{k+1}}$ prepared for the (k+1)th subpattern}
5: **end for**
6: **for all** $u \xrightarrow{z} v \in E_{G_k}$ **do**
7: $s^m_{G_{k+1}}(u \xrightarrow{z} v) := s^m_{G_k}(u \xrightarrow{z} v)$ {Mappings of pattern edges contained by the kth subpattern are copied to the new matching $s^m_{G_{k+1}}$ prepared for the (k+1)th subpattern}
8: **end for**
9: $s^m_{G_{k+1}}(v_{k+1}) := c$ {The (k+1)th pattern node v_{k+1} is now mapped to object c by matching $s^m_{G_{k+1}}$}
10: **for all** $u \xrightarrow{z} v \in E_{G_{k+1}}$ {For all incoming edges of the (k+1)th pattern node v_{k+1}} **do**
11: **for all** $\left\{ a \xrightarrow{e} c \in E_M \mid t(z) = t(e) \wedge s^m_{G_k}(u) = a \right\}$ **do**
12: {Due to the non-existence of parallel edges, only a single link $a \xrightarrow{e} c$ is enumerated here}
13: $s^m_{G_{k+1}}(u \xrightarrow{z} v_{k+1}) := a \xrightarrow{e} c$ {Pattern edge $u \xrightarrow{z} v_{k+1}$ is now mapped to link $a \xrightarrow{e} c$}
14: **end for**
15: **end for**
16: **for all** $v_{k+1} \xrightarrow{z} w \in E_{G_{k+1}}$ {For all outgoing edges of the (k+1)th pattern node v_{k+1}} **do**
17: **for all** $\left\{ c \xrightarrow{e} b \in E_M \mid t(z) = t(e) \wedge s^m_{G_k}(w) = b \right\}$ **do**
18: {Due to the non-existence of parallel edges, only a single link $c \xrightarrow{e} b$ is enumerated here}
19: $s^m_{G_{k+1}}(v_{k+1} \xrightarrow{z} w) := c \xrightarrow{e} b$ {Pattern edge $v_{k+1} \xrightarrow{z} w$ is now mapped to link $c \xrightarrow{e} b$}
20: **end for**
21: **end for**
22: **return** $s_{G_{k+1}}$

Algorithm B.4 The `propagateInsert`(s_{G_k}) method

PROCEDURE `propagateInsert`(s_{G_k})
Require: $G_k \subset G$
1: **if** $\exists u \xrightarrow{z} v_{k+1} \in E_{G_{k+1}}$ **then**
2: {If the (k+1)th pattern node v_{k+1} has at least one incoming edge}
3: **for all** $\left\{ a \xrightarrow{e} b \in E_M \mid t(z) = t(e) \wedge s^m_{G_k}(u) = a \right\}$ **do**
4: `insert`(s_{G_k}, b) {Snapshot s_{G_k} is trying to be extended by mapping the (k+1)th pattern node v_{k+1} to target object b}
5: **end for**
6: **else if** $\exists v_{k+1} \xrightarrow{z} w \in E_{G_{k+1}}$ **then**
7: {If the (k+1)th pattern node v_{k+1} has at least one outgoing edge}
8: **for all** $\left\{ a \xrightarrow{e} b \in E_M \mid t(z) = t(e) \wedge s^m_{G_k}(w) = b \right\}$ **do**
9: `insert`(s_{G_k}, a) {Snapshot s_{G_k} is trying to be extended by mapping the (k+1)th pattern node v_{k+1} to source object a}
10: **end for**
11: **else**
12: {If the (k+1)th pattern node v_{k+1} is unconnected}
13: **for all** $\left\{ c \in V_M \mid t(v_{k+1}) \xleftarrow{*} t(c) \right\}$ **do**
14: `insert`(s_{G_k}, c) {Snapshot s_{G_k} is trying to be extended by mapping the (k+1)th pattern node v_{k+1} to a type conformant object c}
15: **end for**
16: **end if**

Algorithm B.5 The `propagateDelete`(s_{G_k}) method

PROCEDURE `propagateDelete`(s_{G_k})
Require: $G_k \subset G$
1: **for all** $\left\{ s_{G_{k+1}} \in \mathbb{S}^{\mathcal{T}}_G \mid p(s_{G_{k+1}}) = s_{G_k} \right\}$ **do**
2: `delete`($s_{G_{k+1}}$) {Delete all children of snapshot s_{G_k}}
3: **end for**

Algorithm B.6 The `addInsertEntries`(s_{G_k}) method

PROCEDURE `addInsertEntries`(s_{G_k})
Require: $G_k \subset G$
1: {For the (k+1)th pattern node v_{k+1}}
2: $\texttt{INSERT}\left(\left[* \overset{\texttt{type}}{\rightarrow} t(v_{k+1})\right]\right)' := \texttt{INSERT}\left(\left[* \overset{\texttt{type}}{\rightarrow} t(v_{k+1})\right]\right) \cup \{\, s_{G_k} \,\}$
3: **for all** $u \overset{z}{\rightarrow} v_{k+1} \in E_{G_{k+1}}$ {For all incoming edges of the (k+1)th pattern node v_{k+1}} **do**
4: $\quad\texttt{INSERT}\left(\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} *\right]\right)' := \texttt{INSERT}\left(\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} *\right]\right) \cup \{\, s_{G_k} \,\}$
5: **end for**
6: **for all** $v_{k+1} \overset{z}{\rightarrow} w \in E_{G_{k+1}}$ {For all outgoing edges of the (k+1)th pattern node v_{k+1}} **do**
7: $\quad\texttt{INSERT}\left(\left[* \overset{t(z)}{\rightarrow} s^m_{G_k}(w)\right]\right)' := \texttt{INSERT}\left(\left[* \overset{t(z)}{\rightarrow} s^m_{G_k}(w)\right]\right) \cup \{\, s_{G_k} \,\}$
8: **end for**

Algorithm B.7 The `removeInsertEntries`(s_{G_k}) method

PROCEDURE `removeInsertEntries`(s_{G_k})
Require: $G_k \subset G$
1: {For the (k+1)th pattern node v_{k+1}}
2: $\texttt{INSERT}\left(\left[* \overset{\texttt{type}}{\rightarrow} t(v_{k+1})\right]\right)' := \texttt{INSERT}\left(\left[* \overset{\texttt{type}}{\rightarrow} t(v_{k+1})\right]\right) \setminus \{\, s_{G_k} \,\}$
3: **for all** $u \overset{z}{\rightarrow} v_{k+1} \in E_{G_{k+1}}$ {For all incoming edges of the (k+1)th pattern node v_{k+1}} **do**
4: $\quad\texttt{INSERT}\left(\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} *\right]\right)' := \texttt{INSERT}\left(\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} *\right]\right) \setminus \{\, s_{G_k} \,\}$
5: **end for**
6: **for all** $v_{k+1} \overset{z}{\rightarrow} w \in E_{G_{k+1}}$ {For all outgoing edges of the (k+1)th pattern node v_{k+1}} **do**
7: $\quad\texttt{INSERT}\left(\left[* \overset{t(z)}{\rightarrow} s^m_{G_k}(w)\right]\right)' := \texttt{INSERT}\left(\left[* \overset{t(z)}{\rightarrow} s^m_{G_k}(w)\right]\right) \setminus \{\, s_{G_k} \,\}$
8: **end for**

Algorithm B.8 The `addDeleteEntries`(s_{G_k}) method

PROCEDURE `addDeleteEntries`(s_{G_k})
Require: $G_0 \subset G_k$
1: {For the kth pattern node v_k}
2: $\texttt{DELETE}\left(\left[s^m_{G_k}(v_k) \overset{\texttt{type}}{\rightarrow} t(v_k)\right]\right)' := \texttt{DELETE}\left(\left[s^m_{G_k}(v_k) \overset{\texttt{type}}{\rightarrow} t(v_k)\right]\right) \cup \{\, s_{G_k} \,\}$
3: **for all** $u \overset{z}{\rightarrow} v_k \in E_{G_k}$ {For all incoming edges of the kth pattern node v_k} **do**
4: $\quad\texttt{DELETE}\left(\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} s^m_{G_k}(v_k)\right]\right)' := \texttt{DELETE}\left(\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} s^m_{G_k}(v_k)\right]\right) \cup \{\, s_{G_k} \,\}$
5: **end for**
6: **for all** $v_k \overset{z}{\rightarrow} w \in E_{G_k}$ {For all outgoing edges of the kth pattern node v_k} **do**
7: $\quad\texttt{DELETE}\left(\left[s^m_{G_k}(v_k) \overset{t(z)}{\rightarrow} s^m_{G_k}(w)\right]\right)' := \texttt{DELETE}\left(\left[s^m_{G_k}(v_k) \overset{t(z)}{\rightarrow} s^m_{G_k}(w)\right]\right) \cup \{\, s_{G_k} \,\}$
8: **end for**

Algorithm B.9 The `removeDeleteEntries`(s_{G_k}) method

PROCEDURE `removeDeleteEntries`(s_{G_k})
Require: $G_0 \subset G_k$
1: {For the kth pattern node v_k}
2: $\texttt{DELETE}\left(\left[s^m_{G_k}(v_k) \overset{\texttt{type}}{\rightarrow} t(v_k)\right]\right)' := \texttt{DELETE}\left(\left[s^m_{G_k}(v_k) \overset{\texttt{type}}{\rightarrow} t(v_k)\right]\right) \setminus \{\, s_{G_k} \,\}$
3: **for all** $u \overset{z}{\rightarrow} v_k \in E_{G_k}$ {For all incoming edges of the kth pattern node v_k} **do**
4: $\quad \texttt{DELETE}\left(\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} s^m_{G_k}(v_k)\right]\right)' := \texttt{DELETE}\left(\left[s^m_{G_k}(u) \overset{t(z)}{\rightarrow} s^m_{G_k}(v_k)\right]\right) \setminus \{\, s_{G_k} \,\}$
5: **end for**
6: **for all** $v_k \overset{z}{\rightarrow} w \in E_{G_k}$ {For all outgoing edges of the kth pattern node v_k} **do**
7: $\quad \texttt{DELETE}\left(\left[s^m_{G_k}(v_k) \overset{t(z)}{\rightarrow} s^m_{G_k}(w)\right]\right)' := \texttt{DELETE}\left(\left[s^m_{G_k}(v_k) \overset{t(z)}{\rightarrow} s^m_{G_k}(w)\right]\right) \setminus \{\, s_{G_k} \,\}$
8: **end for**

Algorithm B.10 The `checkGraphMorphism`(s_{G_k}, c) method

PROCEDURE `checkGraphMorphism`(s_{G_k}, c)
Require: $G_k \subset G$ {Assert that $s^m_{G_k}$ is a matching for the proper subpattern G_k, and thus, it is only a partial matching for pattern G}
1: **if** $\neg t(v_{k+1}) \overset{*}{\leftarrow} t(c)$ **then**
2: $\quad$ **return** false {Return false, if object c does not conform to the (k+1)th pattern node v_{k+1}.}
3: **end if**
4: **for all** $u \overset{z}{\rightarrow} v_{k+1} \in E_{G_{k+1}}$ {For all incoming edges of the (k+1)th pattern node v_{k+1}} **do**
5: $\quad$ **if** $\nexists a \overset{e}{\rightarrow} c \in E_M \,:\, t(z) = t(e) \wedge s^m_{G_k}(u) = a$ **then**
6: $\quad\quad$ **return** false {Return false, if no links $a \overset{e}{\rightarrow} c$ exist in model M, which could be matched to pattern edge $u \overset{z}{\rightarrow} v_{k+1}$}
7: $\quad$ **end if**
8: **end for**
9: **for all** $v_{k+1} \overset{z}{\rightarrow} w \in E_{G_{k+1}}$ {For all outgoing edges of the (k+1)th pattern node v_{k+1}} **do**
10: $\quad$ **if** $\nexists c \overset{e}{\rightarrow} b \in E_M \,:\, t(z) = t(e) \wedge s^m_{G_k}(w) = b$ **then**
11: $\quad\quad$ **return** false {Return false, if no links $c \overset{e}{\rightarrow} b$ exist in model M, which could be matched to pattern edge $v_{k+1} \overset{z}{\rightarrow} w$}
12: $\quad$ **end if**
13: **end for**
14: **return** true {Returns true as matching $s^m_{G_k}$ together with the mapping that assigns object c to the (k+1)th pattern node v_{k+1} can be a matching for the (k+1)th subpattern G_{k+1}}

Bibliography

[1] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph transformations on domain-specific models. Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University, November 2003.

[2] IBM Alphaworks. Model transformation framework, 2004. `http://www.alphaworks.ibm.com/tech/mtf/`.

[3] Marc Andries and Gregor Engels. A hybrid query language for the extended entity relationship model. *Journal of Visual Languages and Computing*, 8(1), 1997.

[4] Aonix. Ameos framework. http://www.aonix.com/ameos.html.

[5] Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[6] András Balogh and Dániel Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Proc. of the 21st ACM Symposium on Applied Computing*, pages 1280–1287, Dijon, France, April 2006. ACM Press.

[7] András Balogh, Gergely Varró, Dániel Varró, and András Pataricza. Compiling model transformations to EJB3-specific transformer plugins. In *Proc. of the 21st ACM Symposium on Applied Computing*, pages 1288–1295, Dijon, France, April 2006.

[8] Roswitha Bardohl, Karsten Ehrig, Claudia Ermel, Anilda Qemali, and Ingo Weinhold. Specifying visual languages with GenGED. In *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation*, pages 71–81, Grenoble, France, April 2002.

[9] Roswitha Bardohl, Mark Minas, Gabriele Taentzer, and Andy Schürr. *In [38]*, chapter Application of Graph Transformation to Visual Languages, pages 105–180. World Scientific, 1999.

[10] Gernot Veit Batz. Graphersetzung für eine Zwischendarstellung im übersetzenbau. Master's thesis, Universität Karlsruhe, 2005.

[11] Gernot Veit Batz. An optimization technique for subgraph matching strategies. Technical Report 2006-7, Universität Karlsruhe, April 2006.

[12] Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. A first experimental evaluation of search plan driven graph pattern matching. In Manfred Nagl and Andy Schürr, editors, *Proc. of the 3rd International Workshop on the Applications of Graph Transformation with Industrial Relevance*, Kassel, Germany, October 2007.

[13] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, May 2005.

[14] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *Online Proc. of the 2nd OOPSLA Workshop on Generative Techniques in the Context of MDA*, 2003. `http://www.softmetaware.com/oopsla2003/bezivin.pdf`.

[15] Lars Birkedal, Troels Christoffer Damgaard, Arne John Glenstrup, and Robin Milner. Matching of bigraphs. Technical Report TR-2006-88, IT University of Copenhagen, June 2006.

[16] Dorothea Blostein. Application of graph rewriting to document image analysis. In *Proc. of the 6th International Workshop on Theory and Applications of Graph Transformations*, Paderborn, Germany, November 1998.

[17] Boris Böhlen. Specific graph models and their mappings to a common model. In *Proc of the 2nd International Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE)*, volume 3062 of *LNCS*, pages 45–60. Springer-Verlag, September 2003.

[18] Paul Boocock. *Jamda: The Java Model Driven Architecture*, May 2003. `http://sourceforge.net/projects/jamda/`.

[19] Adam Borkowski, Ewa Grabska, and Janusz Szuba. On graph-based knowledge representation in design. In *Computing in Civil Engineering*, pages 1–10, 2002.

[20] David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In *Proc. of the 17th International Conference on Very Large Data Bases (VLDB)*, pages 287–296, 1991.

[21] Peter Braun and Frank Marschall. BOTL the bidirectional object oriented transformation language. Technical Report TUM-I0307, Technische Universität München, 2003.

[22] Horst Bunke, Thomas 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.

[23] Fabian Büttner and Martin Gogolla. Realizing graph transformations by pre- and postconditions and command sequences. In Leila Ribeiro and Ugo Montanari, editors, *Proc. of the 3rd International Conference on Graph Transformation*, volume 4178 of *Lecture Notes in Computer Science*, pages 398–413, Natal, Rio Grande do Norte, Brazil, September 2006. Springer.

[24] David Carlson. *Modeling XML Applications with UML: Practical e-Business Applications*. Addison Wesley Professional, 2001.

[25] Li Chen, Amarnath Gupta, and Mevlüt Erdem Kurul. Efficient algorithms for pattern matching on directed acyclic graphs. In *Proc. of the 21st International Conference on Data Engineering*, pages 384–385, 2005.

[26] Yoeng-Jin Chu and Tseng-Hong Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[27] Edgar Frank Codd. A relational model for large shared data bank. *Communications of the ACM*, 13(6):377–387, June 1970.

[28] Paul R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, Massachusetts, 1995.

[29] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, October 2004.

[30] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Online Proc. of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Anaheim, California, USA, October 2003. `http://www.softmetaware.com/oopsla03/czarnecki.pdf`.

[31] Juan de Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *Fundamental Approaches to Software Engineering: 5th International Conference, FASE 2002*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188, Grenoble, France, April 2002. Springer-Verlag.

[32] Heiko Dörr. *Efficient Graph Rewriting and Its Implementation*, volume 922 of *LNCS*. Springer-Verlag, 1995.

[33] Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels van Eetvelde. Adaptive star grammars. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Proc. of the 3rd International Conference on Graph Transformation*, volume 4178 of *Lecture Notes in Computer Science*, pages 77–91, Natal, Brazil, September 2006. Springer Verlag.

[34] Jack Edmonds. Optimum branchings. *Journal Research of the National Bureau of Standards*, pages 233–240, 1967.

[35] Hartmut Ehrig. *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73, chapter Introduction to the Algebraic Theory of Graph Grammars (A Survey), pages 1–69. Springer Verlag, 1979.

[36] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *Proc. of 8th International Conference on Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63, Edinburgh, United Kingdom, April 2005.

[37] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer Verlag, 2006.

[38] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

[39] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *Proc. of the 14th Annual Symposium on Switching and Automata Theory*, pages 167–180. IEEE, October 1973.

[40] Karsten Ehrig, Esther Guerra, Juan de Lara, László Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *Proc. of the International Workshop on Model Transformation in Practice*, 2005.

[41] Hans-Erik Eriksson and Magnus Penker. *Business Modeling with UML: Business Patterns at Work*. John Wiley & Sons, Inc., February 2000.

[42] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. *In [38]*, chapter The AGG-Approach: Language and Tool Environment, pages 551–603. World Scientific, 1999.

[43] Kerstin Falkowski. Modelltransformationsansätze im Kontext Modellgetriebener Softwareentwicklung. Master's thesis, University of Koblenz-Landau, November 2005.

[44] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In Gregor Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, 1998.

[45] Charles L. Forgy. RETE: A fast algorithm for the many pattern/many object match problem. *Artificial Intelligence*, 19:17–37, 1982.

[46] James Jianghai Fu. Directed graph pattern matching and topological embedding. *Journal of Algorithms*, 22:372–391, 1997.

[47] Christian Fuss, Christof Mosler, Ulrike Ranger, and Erhard Schultchen. The jury is still out: A comparison of AGG, Fujaba, and PROGRES. In Karsten Ehrig and Holger Giese, editors, *Proc. of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques*, volume 6 of *Electronic Communications of the EASST*, pages 183–195, Braga, Portugal, 2007.

[48] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1995.

[49] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[50] Leif Geiger, Christian Schneider, and Carsten Reckord. Template- and modelbased code generation for MDA-tools. In Holger Giese and Albert Zündorf, editors, *Proc. of the 3rd International Fujaba Days*, pages 57–62, Paderborn, Germany, September 2005. `ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-05-259.pdf`.

[51] Rubino Geiß, Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. of the 3rd International Conference on Graph Transformation*, pages 383–397, Natal, Brazil, September 2006. Springer Verlag.

[52] Rubino Geiß and Moritz Kroll. On improvements of the Varró benchmark for graph transformation tools. Technical Report 2007-7, Universität Karlsruhe, IPD Goos, July 2007.

[53] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. of the 1st International Conference on Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105, Barcelona, Spain, October 2002. Springer.

[54] Herbert Göttler. *Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, chapter Attributed Graph Grammars for Graphics, pages 130–142. Springer Verlag, 1983.

[55] Graph transformation benchmarks. Webpage. `http://www.cs.bme.hu/~gervarro/benchmark/`.

[56] Arvind Gupta and Naomi Nishimura. Characterizing the complexity of subgraph isomorphism for graphs of bounded path-width. In *Proc. of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *Lecture Notes in Computer Science*, pages 453–464. Springer Verlag, 1997.

[57] Ashish Gupta, Inderpal Singh Mumick, and Venkatramana S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Proceedings*, pages 157–166, Washington, D.C., USA, 1993.

[58] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[59] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 321–335, Genova, Italy, October 2006. Springer.

[60] Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. In *Fundamental Approaches to Software Engineering: First International Conference, FASE 1998*, volume 1382 of *LNCS*, pages 138–153. Springer, 1998.

[61] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. of the 1st International Conference on Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176, Barcelona, Spain, October 2002. Springer. `http://tfs.cs.tu-berlin.de/~gabi/gHKT02b.pdf`.

[62] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph rewriting – a constructive approach. In Andrea Corradini and Ugo Montanari, editors, *Proc. of Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *Electronic Notes in Theoretical Computer Science*, pages 118–126, Volterra, Pisa, Italy, August 1995. Elsevier.

[63] Ákos Horváth, Dániel Varró, and Gergely Varró. Automatic generation of platform-specific transformation. *Híradástechnika*, 2006.

[64] Ákos Horváth, Gergely Varró, and Dániel Varró. Generic search plans for matching advanced graph patterns. In Karsten Ehrig and Holger Giese, editors, *Proc. of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques*, volume 6 of *Electronic Communications of the EASST*, pages 57–68, Braga, Portugal, March 2007.

[65] HSQLDB. `http://hsqldb.org/`.

[66] Scott E. Hudson. Incremental attribute evaluation: an algorithm for lazy evaluation in graphs. Technical Report 87-20, University of Arizona, 1987.

[67] Jens H. Jahnke, Wilhelm Schäfer, Jörg P. Wadsack, and Albert Zündorf. Supporting iterations in exploratory database reengineering processes. *Science of Computer Programming*, 45(2-3):99–136, 2002.

[68] Rostam Joobbani and Daniel P. Siewiorek. WEAVER: A knowledge-based routing expert. In Hillel Ofek and Lawrence A. O'Neill, editors, *Proc. of the 22nd ACM/IEEE Conference on Design Automation*, pages 266–272, Las Vegas, Nevada, USA, 1985. ACM.

[69] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language MOLA. In *Proc. of Model Driven Architecture: Foundations and Applications*, pages 14–28, Linköping, Sweden, June 2004.

[70] Audris Kalnins, Edgars Celms, and Agris Sostaks. Model transformation approach based on MOLA. In Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt, editors, *Proc. of the International Workshop on Model Transformation in Practice (MTiP 2005)*, October 2005. `http://sosym.dcs.kcl.ac.uk/events/mtip05/`.

[71] Audris Kalnins, Edgars Celms, and Agris Sostaks. Simple and efficient implementation of pattern matching in MOLA tool. In *Proc. of the 7th International Baltic Conference on Databases and Information Systems*, pages 159–167, Vilnius, Lithuania, July 2006.

[72] Martin Karlsch. A model-driven framework for domain specific languages demonstrated on a test automation language. Master's thesis, Hasso-Plattner Institute of Software Engineering Systems, 2007.

[73] Jörg Kiegeland and Hajo Eichler. Enabling comprehensive tool support for QVT. Eclipse Summit Europe, October 2007.

[74] Gerald Kiernan, Christophe de Maindreville, and Eric Simon. Making deductive databases a practical technology: A step forward. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proc. of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 237–246, Atlantic City, New Jersey, USA, May 1990. ACM Press.

[75] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. Design and evaluation of GRAS, a graph-oriented database system for engineering applications. In Hing-Yan Lee, Thomas F. Reid, and Stan Jarzabek, editors, *Proc. of the 6th International Workshop on Computer-Aided Software Engineering*, pages 272–286. IEEE Computer Society Press, 1993.

[76] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph-oriented database system for (software) engineering applications. *Information Systems*, 20(1):21–25, 1995.

[77] Thomas Klein, Ulrich Nickel, Jörg Niere, and Albert Zündorf. From UML to Java and back again. Technical report, University of Paderborn, 2000.

[78] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, April 2003.

[79] Barbara König and Vitali Kozioura. AUGUR2 – a new version of a tool for the analysis of graph transformation systems. In Roberto Bruni and Dániel Varró, editors, *Proc. of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques*, pages 195–204, Vienna, Austria, April 2006.

[80] Alexander Königs and Andy Schürr. MDI: A rule-based multi-document and tool integration approach. *Software and Systems Modeling*, 5(4):349–368, December 2006.

[81] Javier Larrosa and Gabriel Valiente. Graph pattern matching using constraint satisfaction. In *Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 189–196, April 2000.

[82] Michael Lawley and Jim Steel. Practical declarative model transformation with Tefkat. In Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt, editors, *Proc. of the International Workshop on Model Transformation in Practice (MTiP 2005)*, October 2005. `http://sosym.dcs.kcl.ac.uk/events/mtip05/`.

[83] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The Generic Modeling Environment. In *Proc. of the IEEE International Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001.

[84] László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Constraint validation in model compilers. *Journal of Object Technology*, 5(4):107–127, 2006.

[85] László Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. Model transformation with a visual control flow language. *International Journal of Computer Science*, 1(1):45–53, 2006.

[86] Kevin Leopold. Tool support for Model Driven Development: Both commercial products and scientific prototypes, 2007. `http://seal.ifi.uzh.ch/fileadmin/User_Filemount/Vorlesungs_Folien/Seminar_SE/SS07/SemSE07-Kevin_Leopold.pdf`.

[87] Tihamér Levendovszky, Ulrike Prange, and Hartmut Ehrig. Termination criteria for DPO transformations with injective matches. In Arend Rensink, Reiko Heckel, and Barbara König, editors, *Proceedings of the Workshop on Graph Transformation for Concurrency and Verification (GT-VC 2006)*, volume 175 of *Electronic Notes in Theoretical Computer Science*, pages 87–100, Bonn, Germany, August 2006. Elsevier.

[88] Tomas Lillqvist. Subgraph matching in model driven engineering. Master's thesis, Åbo Akademi University, March 2006.

[89] Andrzej Lingas. Subgraph isomorphism for biconnected outerplanar graphs in cubic time. *Theoretical Computer Science*, 63:295–302, 1989.

[90] Andrzej Lingas and Maciej M. Sysło. A polynomial-time algorithm for subgraph isomorphism of two-connected series-parallel graphs. In *Proc. of the 15th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 394–409. Springer Verlag, 1988.

[91] Frank Marschall and Peter Braun. Model transformations for the MDA with BOTL. Technical report, University of Twente, 2003.

[92] David W. Matula. Subtree isomorphism in O($n^{5/2}$). *Annals of Discrete Mathematics*, 2:91–106, 1978.

[93] Alexander Matzner, Mark Minas, and Axel Schulte. Efficient graph matching with application to cognitive automation. In Manfred Nagl and Andy 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] Claudia Meitinger and Axel Schulte. Human-centered automation for UAV guidance: Oxymoron of tautology? the potential of cognitive and co-operative systems. In *Proc. of the 1st Moving Autonomy Forward Conference*, Grantham, United Kingdom, 2006.

[95] Bruno T. Messmer and Horst Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):307–323, 2000.

[96] Mark Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.

[97] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, 2000.

[98] Manfred Münch. *Generic Modelling with Graph Rewriting Systems*. PhD thesis, RWTH Aachen, Aachen, Germany, 2003.

[99] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proc. of the 22nd International Conference on Software Engineering*, pages 742–745. ACM Press, 2000.

[100] Object Management Group. *Common Warehouse Metamodel (CWM) Version 1.0*, February 2001. `http://www.omg.org/technology/cwm/`.

[101] Object Management Group. *Model Driven Architecture — A Technical Perspective*, September 2001. `http://www.omg.org`.

[102] Object Management Group. *Object Constraint Language Specification (in UML 1.4)*, 2001. `http://www.omg.org`.

[103] Object Management Group. *Meta-Object Facility Version 2.0*, April 2003. `http://www.omg.org/mof/`.

[104] Object Management Group. *Common Object Request Broker Architecture: Core Specification, Version 3.0.3*, March 2004. `http://www.omg.org/technology/documents/formal/corba_iiop.htm`.

[105] Object Management Group. *Meta-Object Facility (MOF) 2.0 XMI Mapping Specification Version 2.1*, September 2005. `http://www.omg.org`.

[106] Oracle Corporation. Oracle and TimesTen. http://www.oracle.com/timesten/.

[107] Oracle Corporation. Oracle Database SQL Language Reference, 11g Release 1 (11.1), September 2007. http://www.oracle.com/technology/documentation/database.html.

[108] John Poole, Dan Chang, Douglas Tolbert, and David Mellor. *Common Warehouse Metamodel*. John Wiley & Sons, Inc., 2002.

[109] QVT Partners. *Revised submission for MOF 2.0 Query/Views/Transformations RFP*, August 2003. http://qvtp.org/.

[110] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2002.

[111] Ulrike Ranger and Mathias Lüstraeten. Search trees for distributed graph transformation systems. In Gabor Karsai and Gabriele Taentzer, editors, *Proc. of the 2nd International Workshop on Graph and Model Transformation*, volume 4 of *Electronic Communications of the EASST*, Brighton, United Kingdom, September 2006.

[112] István Ráth and Dániel Varró. Challenges for advanced domain-specific modeling frameworks. In *Proc. of the 1st ECOOP Workshop on Domain-Specific Program Development*, Nantes, France, July 2006.

[113] Arend Rensink. The GROOVE simulator: A tool for state space generation. In John L. Pfalz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[114] Arend Rensink. Representing first-order logic using graphs. In Francesco Parisi-Presicce and Gregor Engels, editors, *Proc. of the 2nd International Conference on Graph Transformation*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335, Rome, Italy, September 2004. Springer.

[115] Arend Rensink. Time and space issues in the generation of graph transition systems. In Tom Mens, Andy Schürr, and Gabriele Taentzer, editors, *Proc. of the International Workshop on Graph-Based Tools*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 127–139, Rome, Italy, October 2004. Elsevier. http://tfs.cs.tu-berlin.de/grabats/.

[116] Arend Rensink and Ronald Nederpel. Graph transformation semantics for a QVT language. In Roberto Bruni and Dániel Varró, editors, *Proc. of the 5th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, ENTCS, pages 45–56, Vienna, Austria, April 2006. Elsevier.

[117] Steven W. Reyner. An analysis of a good algorithm for the subtree problem. *SIAM Journal of Computing*, 6:730–732, 1977.

[118] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[119] Michael Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformations*, volume 1764. Springer, 2000.

[120] Andy Schürr. Specification of graph translators with triple graph grammars. In *Proc. of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. Springer Verlag, 1995.

[121] Andy Schürr. *In [118]*, chapter Programmed Graph Replacement Systems, pages 479–546. World Scientific, 1997.

[122] Andy Schürr, Andreas J. Winter, and Albert Zündorf. *In [38]*, chapter The PROGRES Approach: Language and Environment, pages 487–550. World Scientific, 1999.

[123] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proc. of the 25th Int. Conference on Software Engineering*, pages 74–83, Portland, Oregon, USA, May 2003.

[124] SQLite. `http://www.sqlite.org/`.

[125] Jon Stephens and Chad Russell. *Beginning MySQL Database Design and Optimization: From Novice to Professional*. Apress, October 2004.

[126] Dave D. Straube and M. Tamer Özsu. Query optimization and execution plan generation in object-oriented data management systems. *Knowledge and Data Engineering*, 7(2):210–227, 1995.

[127] Sun Microsystems. *Java 2 Platform Enterprise Edition 5.0*. `http://java.sun.com/javaee/`.

[128] Sun Microsystems. *Java 2 Platform Standard Edition 5.0*. `http://java.sun.com/j2se/1.5.0/`.

[129] Sun Microsystems. *Java Metadata Interface*. `http://java.sun.com/products/jmi/reference/index.html`.

[130] Sun Microsystems. *JSR 220: Enterprise JavaBeans, Version 3.0*, May 2006. `http://java.sun.com/products/ejb/docs.html`.

[131] Maciej M. Sysło. The subgraph isomorphism problem for outerplanar graphs. *Theoretical Computer Science*, 17:91–97, 1982.

[132] Adam Szalkowski. Negative Anwendungsbedingungen für das suchprogrammbasierte Backend von GrGen. Master's thesis, Universität Karlsruhe, October 2005.

[133] Gabriele Taentzer, Enrico Biermann, Dénes Bisztray, Bernd Bohnet, Iovka Boneva, Artur Boronat, Leif Geiger, Rubino Geiß, Ákos Horváth, Ole Kniemeyer, Tom Mens, Benjamin Ness, Detlef Plump, and Tamás Vajk. Generation of Sierpinski triangles: A case study for graph transformation tools. In *Proc. of the 3rd International Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, Lecture Notes in Computer Science. Springer, 2008.

[134] Gabriele Taentzer and Arend Rensink. Ensuring structural constraints in graph-based models with type inheritance. In Maura Cerioli, editor, *Proc. 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *LNCS*, pages 64–79. Springer Verlag, 2005.

[135] Naveed Ahsan Tariq and Naeem Akhter. Comparison of Model Driven Architecture (MDA) based tools. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, June 2005.

[136] The Eclipse Foundation. *Eclipse Modeling Framework*. `http://www.eclipse.org/modeling/emf/`.

[137] Transaction Processing Performance Council. *TPC Benchmark C (Standard Specification, Revision 5.3)*, April 2004. `http://www.tpc.org/tpcc/`.

[138] Laurence Tratt. A change propagating model transformation language. Technical Report TR-06-07, King's College London, August 2006.

[139] U2-Partners. *UML: Infrastructure v. 2.0 (Third revised proposal)*, January 2003. `http://www.u2-partners.org/artifacts.htm`.

[140] Julian R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.

[141] Gabriel Valiente and Conrado Martínez. An algorithm for graph pattern-matching. In *Proc. of the 4th South American Workshop on String Processing*, volume 8 of *International Informatics Series*, pages 180–197, 1997.

[142] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. In Hartmut Ehrig and Gabriele Taentzer, editors, *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 14–21, Berlin, Germany, March 2000.

[143] Dániel Varró, Gergely Varró, and András Pataricza. Visual graph transformation in system verification. In Elena Gramatova, Hans Manhaeve, and Adam Pawlak, editors, *DDECS 2000 Design and Diagnostics of Electronic Circuits and Systems*, pages 137–141, Bratislava, Slovakia, April 2000.

[144] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

[145] Gergely Varró. Incremental graph transformation in relational databases. In *Proc. of the 4th Conference of PhD Students in Computer Science*, page 124, Szeged, Hungary, July 2004.

[146] Gergely Varró. Towards incremental graph transformation in Fujaba. In Holger Giese, Andy Schürr, and Albert Zündorf, editors, *Proc. of the 2nd International Fujaba Days*, pages 3–6, Darmstadt, Germany, September 2004.

[147] Gergely Varró. Gráftranszformációs benchmarkok jellemzése. In Enikő Bitay, editor, *XI. Fiatal Műszakiak Tudományos Ülésszaka*, pages 379–382, Cluj, Romania, March 2006. In Hungarian.

[148] Gergely Varró. Practical issues in the implementation of graph pattern matching. In *Proc. of the 5th Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, pages 191–200, Sendai, Japan, April 2007.

[149] Gergely Varró. Implementing an EJB3-specific graph transformation plugin by using database independent queries. In Roberto Bruni and Dániel Varró, editors, *Proc. of the 5th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, volume 211 of *Electronic Notes in Theoretical Computer Science*, pages 121–132, Vienna, Austria, April 2008. Elsevier.

[150] Gergely Varró, Katalin Friedl, and Dániel Varró. Graph transformation in relational databases. In Tom Mens, Andy Schürr, and Gabriele Taentzer, editors, *Proc. of the International Workshop on Graph-Based Tools (GraBaTs 2004)*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 167–180, Rome, Italy, October 2004. Elsevier.

[151] Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a graph transformation engine in relational databases. *Software and Systems Modeling*, 5(3):313–341, September 2006.

[152] Gergely Varró, Ákos Horváth, and Dániel Varró. Recursive graph pattern matching with magic sets and global search plans. In *Proc. of the 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance*, 2007.

[153] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. Technical Report TUB-TR-05-EE17, Budapest University of Technology and Economics, March 2005. `http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf`.

[154] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 79–88, Dallas, Texas, USA, September 2005. IEEE Computer Society Press.

[155] Gergely Varró and Dániel Varró. Graph transformation with incremental updates. In Reiko Heckel, editor, *Proc. of the 4th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2004)*, volume 109 of *Electronic Notes in Theoretical Computer Science*, pages 71–83, Barcelona, Spain, December 2004. Elsevier.

[156] Gergely Varró, Dániel Varró, and Katalin Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In Gabor Karsai and Gabriele Taentzer, editors, *Proc. of International Workshop on Graph and Model Transformation (GraMoT'05)*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 191–205, Tallinn, Estonia, September 2005.

[157] Gergely Varró, Dániel Varró, and Andy Schürr. Incremental graph pattern matching: Data structures and initial experiments. In Gabor Karsai and Gabriele Taentzer, editors, *Proc. of the 2nd International Workshop on Graph and Model Transformation*, volume 4 of *Electronic Communications of the EASST*, Brighton, United Kingdom, September 2006.

[158] Attila Vizhanyo, Aditya Agrawal, and Feng Shi. Towards generation of efficient transformations. In *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.

[159] W3C. *XSL Transformations (XSLT) Version 1.0*, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[160] Xcelerix. http://www.xcelerixtech.com/.

[161] Ke-Bing Zhang, Mehmet A. Orgun, and Kang Zhang. Visual language semantics specification in the VisPro system. In *Proc. of the Pan-Sydney Area Workshop on Visual Information Processing*, pages 121–127, Sydney, Australia, 2002. Australian Computer Society.

[162] Chunying Zhao, Jun Kong, and Kang Zhang. Design pattern evolution and verification using graph transformation. In *Proc. of the 40th Hawaii International Conference on System Sciences*, Waikoloa, Hawaii, USA, January 2007.

[163] Albert 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.

[164] Albert Zündorf. *Rigorous Object Oriented Software Development*. Habilitation thesis, University of Paderborn, 2001.