Incremental Model Queries in Model-Driven Design

PhD Thesis

Gábor Bergmann

MSc in Technical Informatics

Supervisor:
Dr. Dániel Varró, DSc
associate professor

Budapest, July 2013
Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Bergmann Gábor 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.

Declaration of own work and references

I, Gábor Bergmann, hereby declare, that this thesis, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Budapest, 2013. 07. 11.

Bergmann Gábor
Acknowledgements

First and foremost, I would like to thank Dr. Dániel Varró. As my advisor, he has continuously provided me with guidance, valuable feedback, and interesting challenges to solve.

I would have never gotten this far without the team of colleagues here at the Fault Tolerant Systems Research Group, Budapest University of Technology and Economics. I am grateful to Prof. Dr. András Pataricza and Dr. István Majzik, for providing financial and other kinds of support for my research during their leadership of the group. Dr. István Ráth, Ákos Horváth, Ábel Hegedűs, Zoltán Ujhelyi, Benedek Izsó, along with past collaborators András Ökrös, Dr. Gergely Varró, and others, deserve my thanks for being excellent team players, co-authors, co-developers, professional role models, and often a sounding board for my ideas. I had the great opportunity to build upon their work, and I find it a very satisfying feeling whenever I see that they are also benefit from my results.

As pursuing the PhD required a lot of work and effort, with many harsh deadlines along the way, I have amassed a great debt towards my family and my girlfriend by not always being able to spend enough time with them. I have to thank them for showing great tolerance, as well as invaluable support.

This research was realized in the frames of TÁMOP 4.2.4. A/1-11-1-2012-0001 „National Excellence Program – Elaborating and operating an inland student and researcher personal support system”. The project was subsidized by the European Union and co-financed by the European Social Fund. The research was also partially supported by the Hungarian grant project TÁMOP 4.2.1. B-09/1/KMR-2010-0002, the Hungarian CERTIMOT (ERC-HU-09) project, the Hungarian–French Intergovernmental S&T Cooperation Program under the ”Methods and algorithms to enhance the dependability of services in sensor networks” (F-Egide-PHC-19476SH, FR 6/2008), as well as EU projects SENSORIA (IST-3-016004) FP6 and SecureChange (ICT-FET-231101) FP7. My publishing was partially supported by the Schnell László Foundation.

Besides the financial support, some of these projects have also provided me with the opportunity for fruitful cooperation with excellent international colleagues who are too numerous to list, many of whom are now also my co-authors. I am extending my thanks to them as well.
Summary

The discipline of model-driven engineering (MDE) is gaining more acceptance in several areas of software and system engineering as it delivers higher-quality products in a shorter development lifecycle. MDE redefines the engineering process as driven by the creation, revision, communication, analysis and automated derivation of formal models of system structure and behavior.

Years of extensive academic research and industrial innovation in modeling technology and language engineering have made the development of domain-specific modeling environments a practical choice. The key contributing factors are efficient and flexible support for creating modeling tools that represent, display, edit, and automatically process models. In the specific application domain of security requirement engineering, for instance, an integrated tool may offer various features in addition to simply modeling security requirements. The requirement model may be processed automatically to check security properties. Additional manual analysis may be carried out to reason about risks and security. Finally, the requirement model may be mapped to other models conforming to different formalisms.

Engineering processes, especially those in software engineering, are typically iterative in nature. Thus, in model-driven engineering, evolving models are to be expected, posing a challenge to modeling tools and model queries in particular. In terms of the security requirement modeling example, when the requirement model is changed during the engineering process, the results of any previously performed analysis and transformation tasks become obsolete, and updating them will require time and other resources.

The main target of research in automated model processing has been the development of various model transformation approaches. The challenge of providing declarative languages and efficient evaluation strategies for model queries has received limited attention, despite being a crucial component of specifying and executing transformations and a valuable building block in other use cases including model validation.

This thesis is centered around providing languages, methods and technologies for model queries and transformations to deal with the evolving nature of models.

As a cornerstone of all contributions in the thesis, (a) I propose an incremental computation strategy for the evaluation of queries over evolving models to mitigate the cost of repeated application of processing steps. This proposal is supported by a detailed formal underpinning and experimental evaluation results. For practical feasibility, (b) I integrate the solution to the modeling technology of Eclipse Modeling Framework (EMF) and validate the approach by performance measurements. Contributions include a query syntax tailored to EMF, and the design of an interface between the execution engine and the model management platform. To lift the concern of evolving models to the level of change-driven transformations, (c) I designed an extended query language that can specifically express the way the model evolves. The language is supplemented by execution strategies specific to the application scenario. To illustrate the practical application of the theoretical contributions of the thesis in the context of a case study, (d) I additionally present a complex integrated environment that supports modeling and analysis of security requirements, taking advantage of the previously proposed techniques.

The results of this thesis form an integral part of the VIATRA2 model transformation framework and the EMF-IncQuery model query technology. The thesis contains a case study from the domain of security requirement engineering investigated in the SecureChange European Union FP7 research project. A second case study is from automotive engineering.
Összefoglaló

Napjainkban a modellezéssel tervezett (MDE) a szoftver- és rendszerfejlesztés számos területén egyre elfogadottabbá válik, mivel rövidebb fejlesztési ciklussal képes emelt minőségű termékek előállítására. Az MDE alapján a mérnöki folyamat középpontjában a rendszer felépítését és viselkedését leíró formális modellek készítése, felülvizsgálata, közlése, elemzése és automatizált származtatása áll.

A modellezési és nyelvertevézési technikák területén évek óta tartó kiterjedt akadémiai kutatás és ipari innováció elérhető választásai mellett az MDE alapján a mérnöki folyamat középpontjában a rendszer felépítését és viselkedését leíró formális modellek készítése, felülvizsgálata, közlése, elemzése és automatizált származtatása áll.

A modellezés alapján a biztonsági követelmények elemzésének szakterületén egy integrált eszköz a biztonsági követelmények modellezésén túl többféle szolgáltatást nyújthat. A követelménymodell automatikus feldolgozásával biztonsági tulajdonságok ellenőrizhetők. A biztonság és a kockázatok további, manuális elemezése is lehetséges. Végül a követelménymodellekeket leképezhető középértéki formalizmusokhoz igazodó modelllekről.

A mérnöki folyamatok jellemzően iteratív termézetűek, különösen a szoftvertervezésben. Így a modellezéssel szoftvertervezés során változó modellekre kell számítani, ami kihívást jelent a modellkezelő eszközök körbe vezetésére, valamint automatizált feldolgozására képes modelllező eszközök létrehozásának rugalmasságát és hatékonyságát támogatása. Például a biztonsági követelmények elemzésénél szakterületén egy integrált eszköz a biztonsági követelmények modellezésén túl többféle szolgáltatást nyújthat. A követelménymodell automatikus feldolgozásával biztonsági tulajdonságok ellenőrizhetők. A biztonság és a kockázatok további, manuális elemezése is lehetséges. Végül a követelménymodellek készítését már formalizmusokhoz igazodó modelllekről.

Az automatizált modellfeldolgozás területén a legtöbb kutatás a modelltranszformációk fejlesztésére irányult. Ezzel szemben a modell-lekérdezések számára biztosított, kevésbé formális nyelvek és hatékon kielégítő érzékelési stratégiai kevésbé hatékony kifejezéseket kaptak, holott a transzformációk specifikációnak és végrehajtásának kritikus összetevőjéről van szó, amely többek között a modellvalidáció fontos építőköve is egyben.

Jelen értekezés központi célja olyan nyelveket, módszereket és technológiákat biztosítani a modell-lekérdezések és modelltranszformációk számára, amelyek megfelelőknek a modellle változó természetével.

Az értekezés legalapvetőbb eredményeként (a) egy inkrementális számítási stratégiát javasoltam a változó modellek feletti lekérdezések kielégülésére, hogy a feldolgozás lépései ismételt alkalmazásának költségét enyhítsen. A javaslatot részletes formális megalapozás és kísérleti eredmények megváltoztatására (b) integráltan a megoldást az Eclipse Modeling Framework (EMF) modelllekedési technológiához, és teljesítménymérésekkel validáltam. Az eredmények közé tartozik egy EMF-hez tervezett lekérdezés szintaxis, továbbá a végrehajtómotor és a modellkezelő platform közötti interfész megtervezése. Annak érdekében, hogy a változó modellek kérdését változásvezérelt transzformációk szintjére elérjük, (c) terveztem egy kiterjesztett lekérdezőnyelvet, amely képes a modell változásának módját kifejezni. A nyelvet az egyes alkalmazási helyzetekre jellemző lekérdezési stratégiák egészítik ki. Az elméleti eredményeim gyakorlati alkalmazhatóságának esetileg annak kísérleti kimutatására (d) kidolgoztam egy komplex integrált környezetet biztonsági követelmények modellezésére és elemzésére, amely a korábban javasolt technikákra épít.

Az értekezés eredményei a ViTAR2 modelltranszformációs keretrendszer és az EMF-INCQUERY modell-lekérdező technológia szerves részét alkotják. Az értekezésben megjelenik a SecureChange EU FP7-es kutatási projekt egyik esetételének a biztonságikövetelmény-modellezés szakterületéről. Egy második esétetelének a gépjármű-elektronika területéről származik.
# Contents

## 1 Introduction
1.1 Model-driven engineering ........................................ 1  
  1.1.1 The paradigm of model-driven engineering ................. 1  
  1.1.2 Model transformation and model queries .................. 2  
  1.1.3 Incremental, live and change-driven transformations .... 3  
  1.1.4 Example application domains .............................. 4  
1.2 Challenges and contributions .................................. 5  
  1.2.1 Use cases of model queries ................................. 5  
  1.2.2 Challenges ................................................. 6  
  1.2.3 Contributions of the thesis ............................... 7  
1.3 The structure of the thesis .................................... 7

## 2 Background
2.1 Modeling preliminaries ......................................... 9  
  2.1.1 Running example: Petri nets .............................. 9  
  2.1.2 Graph models and metamodeling .......................... 10  
  2.1.3 Operations on metamodels ................................. 13  
  2.1.4 Attribute values ......................................... 16  
  2.1.5 Model access operations ................................. 17  
  2.1.6 Modeling paradigms ..................................... 21  
2.2 Graph patterns and graph transformation ..................... 23  
  2.2.1 Graph pattern basics .................................... 23  
  2.2.2 Complex graph patterns ................................. 25  
  2.2.3 Graph pattern matching ................................. 33  
  2.2.4 Graph transformation rules ............................. 34

## 3 Incremental Graph Pattern Matching
3.1 Incremental graph pattern matching basics .................... 37  
  3.1.1 Stateful pattern matching ............................... 37  
  3.1.2 Algorithmic complexity of stateful pattern matching .... 39  
3.2 Related work .................................................... 41  
  3.2.1 Related work: incremental graph pattern matching in graph transformation ... 41  
  3.2.2 Related work: incremental matcher algorithms for production rule systems ... 43  
  3.2.3 Related work: incremental maintenance in databases ................................. 44  
  3.2.4 Related work: incremental maintenance of queries in MDE ... 45  
3.3 Principles of the Rete algorithm .............................. 45  
  3.3.1 High-level overview of components and structure ....... 46
## CONTENTS

3.3.2 High-level overview of operation ........................................... 47  
3.3.3 Formalization ................................................................. 49  
3.3.4 Discussion of algorithmic complexity ................................... 52  
3.4 Adapting Rete for graph pattern matching ................................. 53  
3.4.1 Basic graph pattern matching with Rete ............................... 53  
3.4.2 Rete pattern matching with advanced pattern language features .... 56  
3.4.3 Rete pattern matching with attributes .................................. 61  
3.4.4 Realization considerations .................................................. 64  
3.5 Performance evaluation .......................................................... 67  
3.5.1 Petri net firing: a model simulation benchmark ...................... 67  
3.5.2 Measurement results ......................................................... 68  
3.5.3 Related work in graph transformation benchmarking .............. 70  
3.5.4 Performance discussion ...................................................... 70  
3.6 Chapter conclusions ............................................................... 71

4 Advanced Incremental Pattern Matching ...................................... 73  
4.1 Incremental pattern matching on multi-core platforms .................. 73  
4.1.1 Introduction ................................................................. 73  
4.1.2 Related work ............................................................... 74  
4.1.3 Concurrent pattern matching and model manipulation .............. 75  
4.1.4 Multi-threaded pattern matching with Rete ........................... 77  
4.2 Graph patterns with transitive closure ..................................... 80  
4.2.1 Introduction ................................................................. 80  
4.2.2 The transitive closure problem .......................................... 81  
4.2.3 Integration of transitive closure into Rete ......................... 83  
4.2.4 Incremental graph transitive closure maintenance algorithms .... 84  
4.2.5 Related work ............................................................... 87  
4.3 Incrementality on top of existing relational databases .................. 87  
4.3.1 Motivation ................................................................. 87  
4.3.2 Overview of the approach ............................................... 88  
4.3.3 Basic pattern matching over a relational database .................. 90  
4.3.4 Incrementality using cache tables and triggers .................... 90  
4.3.5 Advanced pattern language features ................................... 92  
4.3.6 Performance observations ............................................... 93  

5 Incremental Model Queries over Industrial EMF Models ................... 95  
5.1 Platform and case study ....................................................... 95  
5.1.1 EMF technical preliminaries ............................................ 95  
5.1.2 Motivating example: security requirements .......................... 97  
5.2 EMF model queries based on graph patterns .............................. 99  
5.2.1 Structural constraints .................................................... 99  
5.2.2 Attribute and arithmetic constraints .................................. 100  
5.2.3 Query language structure ............................................... 102  
5.3 Integrating incremental pattern matching to EMF ....................... 102  
5.3.1 EMF as graph model with elementary queries ...................... 102  
5.3.2 Translating from EMF notifications to graph delta ................ 103  
5.4 Performance analysis of EMF model queries .............................. 105
<table>
<thead>
<tr>
<th>5.4.1</th>
<th>Measurement scenario: constraint checking in AUTOSAR models</th>
<th>105</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.2</td>
<td>Benchmarking</td>
<td>108</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Analysis of the results</td>
<td>109</td>
</tr>
<tr>
<td>5.5</td>
<td>Chapter conclusions</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>Queries and Transformations for Security Requirements</th>
<th>113</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>113</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Overview of the S/e.scCMER tool</td>
<td>114</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Metamodels in the S/e.scCMER tool</td>
<td>115</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Example scenarios from the ATM domain</td>
<td>115</td>
</tr>
<tr>
<td>6.2</td>
<td>Continuous validation of security requirements models</td>
<td>122</td>
</tr>
<tr>
<td>6.3</td>
<td>Change impact analysis on informal arguments</td>
<td>124</td>
</tr>
<tr>
<td>6.4</td>
<td>Bidirectional change-driven requirements synchronization</td>
<td>125</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Properties of the Si* metamodel</td>
<td>125</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Mapping between the languages</td>
<td>126</td>
</tr>
<tr>
<td>6.5</td>
<td>Chapter conclusions</td>
<td>127</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>Queries for Change-driven Transformations</th>
<th>129</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Terminology of change in change-driven transformations</td>
<td>129</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Aspects of change</td>
<td>130</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Transformations of change</td>
<td>132</td>
</tr>
<tr>
<td>7.2</td>
<td>Language for change-driven transformations</td>
<td>134</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Requirements and motivation for change-driven rules</td>
<td>134</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Change patterns</td>
<td>135</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Change-driven rules</td>
<td>140</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Challenges addressed</td>
<td>143</td>
</tr>
<tr>
<td>7.3</td>
<td>Case study: bidirectional synchronization</td>
<td>144</td>
</tr>
<tr>
<td>7.4</td>
<td>Case study: change impact analysis by evolutionary constraints</td>
<td>147</td>
</tr>
<tr>
<td>7.5</td>
<td>Implementation strategies for evaluating graph change patterns</td>
<td>149</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Change query evaluation in documented or invisible change scenarios</td>
<td>149</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Change query evaluation in live change scenarios</td>
<td>151</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Implementing change-driven rules</td>
<td>153</td>
</tr>
<tr>
<td>7.6</td>
<td>Discussion</td>
<td>154</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Theoretical discussion</td>
<td>154</td>
</tr>
<tr>
<td>7.6.2</td>
<td>Expressiveness wrt. model synchronization languages</td>
<td>155</td>
</tr>
<tr>
<td>7.6.3</td>
<td>Practical discussion</td>
<td>156</td>
</tr>
<tr>
<td>7.7</td>
<td>Related work</td>
<td>157</td>
</tr>
<tr>
<td>7.8</td>
<td>Chapter conclusions</td>
<td>160</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>Conclusions</th>
<th>161</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>New scientific results</td>
<td>161</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Efficient, incremental pattern matching in a model-driven environment</td>
<td>161</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Incremental model queries over industrial EMF models</td>
<td>162</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Supporting change-driven transformation specification by queries</td>
<td>163</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Queries and transformation in modeling security requirements</td>
<td>164</td>
</tr>
<tr>
<td>8.2</td>
<td>Future directions</td>
<td>165</td>
</tr>
<tr>
<td>8.3</td>
<td>Applications of new scientific results</td>
<td>166</td>
</tr>
</tbody>
</table>
8.3.1 Incremental pattern matcher module of the VIATRA2 model transformation framework ........................................ 166
8.3.2 EMF-IncQuery .................................................................. 166
8.3.3 SeCMER tool prototype ................................................... 167
List of publications ............................................................... 169

Bibliography ............................................................................. 175
Chapter

1

Introduction

1.1 Model-driven engineering

1.1.1 The paradigm of model-driven engineering

The discipline of model-driven engineering (MDE) is gaining more and more prominence in certain areas of software and system engineering, primarily where faults can lead to human injury or significant damage in property, as it delivers higher-quality products in a shorter development lifecycle (see e.g. [HWR13]). According to MDE, the focus of the engineering process is on creating and analyzing models at different levels of abstraction, and deriving them from other models. These models conform to various modeling languages.

Modeling may start in an early phase of the engineering process, when requirements for the system under design are elicited. In light of the requirements, system design commences with creating high-level abstract models, then producing lower-level models enriched with design decisions and realization considerations after a series of refining steps. The models can be continuously verified in order to identify design faults as soon as possible.

Model Driven Architecture (MDA [OMG01]) by the Object Management Group is one of the MDE-based design approaches with the following characteristics. As the system under design is often required to be realized upon various target platforms corresponding to different technologies, such design processes may involve a platform-independent model (PIM), which encompasses significant application-specific behavioral principles and realization parameters, but technological aspects are not detailed yet. Afterwards, depending on the available technological context, the PIM may be mapped to various platform-specific models (PSM), from which program modules realizing the designed software components can finally be produced (partially automatically).

The concept of models in MDE is vague in the sense that it may even involve differential equations or spatial configurations in certain domains of system engineering. However, models involved in software engineering are essentially labeled graphs, and are typically sparse (i.e. the number of edges is roughly linearly proportional to the number of vertices). The labels applicable for vertices and edges of a modeling language (including types and attributes), along with their rules of interconnection, are defined by the metamodel of the language. Note that only the abstract, formal structure of the model (the so-called abstract syntax) is characterized here as graph-like; while the user-friendly visual depiction of the model (concrete syntax) can independently be of diagram, text, or any other form.

While there are some extensible formalisms intended as a general purpose way of representing
CHAPTER 1. INTRODUCTION

models (such as UML [OMG11], SysML [OMG12b]), industrial practice seems to increasingly prefer domain-specific modeling languages (DSML) instead, which can be tailored to the needs of application domains and actual design processes. However, developing such a DSML (along with its associated tool support) is a cost-intensive task requiring special skills; therefore domain-specific modeling (DSM) technologies have emerged to provide aid. Built on the successful Eclipse platform [ECLb], the Eclipse Modeling Framework (EMF, [EMF]) is a leading DSM technology that is considered a de facto industrial standard. A DSML development process with EMF involves defining a metamodel (using the Ecore formalism), from which several components of the modeling tool can be automatically derived. Numerous generative and generic technologies assist the creation of tool support for EMF-based DSMLs; one can define a textual concrete syntax using a grammar or a visual concrete syntax using graphical elements, while code generators can be created by specifying textual templates for the modeling language.

1.1.2 Model transformation and model queries

Several steps of MDE can be partially or completely automated by model transformation (MT). First to gain wide-spread use was code generation, more precisely, model-to-text transformation (M2T). Code generators map models (such as PSM in case of MDA) to source code artifacts that will run on the target implementation platform. Deployment descriptors, test suites or documentation could be synthetized as well in addition to program code. Model-to-model transformation (M2M) is also gaining importance. A usage example from MDA would be the automated support of PIM-to-PSM mappings, which adds platform-specific knowledge to the PIM. Other kinds of model transformations may include the synchronization between different models representing the same system in different ways, for experts of different domains (e.g. security requirements and threat analysis). Finally, model validation or design rule checking can be thought of as a special case of model transformation, where the output is the detected violations of constraints.

A model transformation program can be implemented using any general-purpose programming language and toolkit. However, there are platforms specifically designed to support the creation of model transformations. There are basically three kinds of help one can expect from an MT system: (i) it can aid the transformation developer in processing the source model, such as in the form of queries, (ii) it can simplify the creation of elements of the target model, (iii) finally it can provide the control flow of the model transformation, managing state and traceability information.

From all these services, my work focuses primarily on investigating declarative model queries. Queries evaluated over model elements play an important role in code generation from PSM models, in M2M transformations for analysis or other purposes, in the simulation of behavioral models, in state space exploration, in report generation, etc. One of the important applications in DSM is automated validation of well-formedness constraints associated with the language. For instance, the AUTOSAR [AUT] standard defines hundreds of such constraints. DSM frameworks may provide an ability to express queries in a query language designed specifically for this task (see e.g. OCL [OMG12a], QVT [OMG08] queries), and evaluate them by a query engine.

The mathematical formalism of graph rewriting or graph transformation (GT [EEKR99, BV06]) provides a declarative, rule-based paradigm that can used, among many other purposes, to specify M2M transformations. The basic building block of GT is the graph pattern, which is essentially a declarative query: it identifies certain parts of the graph model based on structural and other criteria. Model manipulation is expressed in the form of graph transformation rules that consist of two graph patterns, and describe the transformation step where a subgraph matching the LHS pattern is substituted with a subgraph conforming to the RHS pattern. Therefore the GT formalism covers both the
1.1. MODEL-DRIVEN ENGINEERING

Figure 1.1: Roles of model transformations in MDE (inspired by [Pat06])

source model processing and target model manipulation aspects of MT, and in certain cases it does not require an externally defined control flow. Declarative model queries using graph patterns is a core topic of my thesis.

For the special task of model-to-model synchronization, Triple Graph Grammars (TGG, [Sch95, KW07]) and QVT [OMG08] provide an even higher level of abstraction in specifying transformations. TGG is based on GT, and TGG rules can be translated to GT rules. These languages allow very concise definition of common model mapping tasks; e.g. a single TGG rule can immediately be interpreted as a bidirectional, incremental (see Section 1.1.3) synchronization.

The above mentioned variety of roles and application areas of MT in MDE are illustrated on Figure 1.1.

1.1.3 Incremental, live and change-driven transformations

In a model-driven engineering process, models usually do not exist as static, immutable facts, but are rather undergoing constant evolution; implying that any previously conducted model analysis must be re-evaluated, and the effect of changes may propagate to other models as well. This evolution may happen due to requirement changes (potentially as late as years after the delivery of the system), or on a shorter time-scale the creation of ever newer model versions according to agile, iterative development methodologies, or simply the consequence of fixing problems detected by model validation. In fact, model editing actually consists of a sequence of small, atomic manipulation operations; this can also be regarded as the continuous evolution of a model, during which e.g. immediate feedback of model validation would be useful.

Repeatedly processing a (large scale) model after each small change can lead to significant performance issues. It can be more advantageous to apply incremental evaluation techniques [HLR06], taking into account the evolving nature of the model. In certain use cases (e.g. well-formedness checks) incremental queries have a great performance advantage [11][ISR+13].
Source incrementality is the property of a transformation that it only re-evaluates the modified parts of the source model. One of the central topics of my thesis is efficient evaluation of queries against evolving models through providing source incrementality.

Target incrementality, on the other hand, means that only the necessary parts of the target model are modified by the transformation, there is no need to recreate the new target model from scratch. The latter property, beyond direct gains in performance, has the benefit that connections, references between the target model and other external models are left intact and need not be recreated. Moreover, if the target model contains pieces of information (such as platform-specific design decisions in a PSM mapped from PIM) that do not stem from the source model, then the lack of target incrementality would lead to outright information loss.

After model evolution, the traditional MT approach restores the logical correspondence between source and target models by re-executing the transformation (which is efficient in case of source and target incrementality). A live transformation [HLR06], however, is continuously active, immediately reacting to events (changes of the source model) by keeping the target model synchronized. In this case source and target incrementality is highly beneficial.

Change-driven transformations [RVV09] are transformations that process changes of models – more precisely, even their specification is given in terms of consuming changes of the source model and producing changes of the target model. In this sense, source and target incrementality is a prerequisite for them. A transformation specified in a change-driven way can be executed as a live transformation, but this is far from being the only application scenario. It is possible to execute change-driven transformations even in cases where the source and target models of the M2M mapping are not actually available at the same computing resource, and can only communicate through the propagation of change information.

1.1.4 Example application domains
1.1.4.1 Modeling security requirements

Complex systems are typically designed to meet the needs of multiple stakeholders. Requirement models (such as the UML [OMG11] or SysML [OMG12b] standards, or KAOS [LL04]) help the designers obtain an overview of the needs and goals of various stakeholders, i.e. their requirements.

Security is a design concern aiming to avoid damages caused by adversarial persons, including damages to information assets (data security). It should not be confused with the related design concern of safety, which, regardless whether the damage is intentional, attempts to avoid primarily human injury, secondarily disproportionate physical damage to property. The thesis will address some concepts related to security.

System security is a broad area involving diverse design challenges. Software aspects include constructing secure cryptographic algorithms, communication protocols, and techniques for the prevention or detection of weaknesses and vulnerabilities in their implementation. Beyond software, technical aspects involve hardware solutions and also physical security. Finally, social aspects involve training humans involved with the system and establishing appropriate procedures for handling normal business and incidents. However, none of these techniques can be applied unless we know what is there to protect and who should have access; therefore security design must be preceded by gathering security requirements.

Security requirement modeling [M+02, NNY10] is the process of creating and using requirement models that record the security needs and goals of stakeholders. For example, the modeling language Si* [MMZ07] can express trust between stakeholders and the delegation of responsibilities and permis-
sions. The related field of security risk modeling \cite{LSS11} focuses on assessing the impact of potential threats to the system, the vulnerabilities against these threats, the characteristics of attackers that can exploit these vulnerabilities, and their associated risk of doing so.

With the application of model queries and transformations, security requirement models have the potential for conducting analysis that reveals inconsistent security needs, as well as for automatically providing solutions and guidelines for later phases of system design.

### 1.1.4.2 Embedded systems engineering in the automotive industry

As onboard electronic systems in automobiles are embedded systems with high safety requirements, the automotive industry benefits from rigorous methods of software and system engineering, calling for the use of model-driven techniques. In particular, proper analysis of models may lead to detecting design flaws early, which is a significant boost in efficiency. Due to the heterogeneity of software and hardware modules produced by various vendors, their integration is a challenge on the level of design models as well as on the level of implementation.

AUTOSAR (short for Automotive Open System Architecture, \cite{AUT}) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. The objectives of the AUTOSAR partnership include the implementation and standardization of basic system functions while providing a highly customizable platform which continues to encourage competition on innovative functions. The purpose of the common standard is to help the integration of functional modules from multiple suppliers and increase scalability to different vehicle and platform variants. It aims to be prepared for the upcoming technologies and to improve cost-efficiency without making any compromise with respect to quality.

### 1.2 Challenges and contributions

#### 1.2.1 Use cases of model queries

While declarative specification and execution of model transformations have now been widely studied and regarded as a significant field of research (see for instance the series International Conference on Model Transformations \cite{ICM13}), the enabling support technology of declarative model queries deserves its own research focus. Use cases of model queries include the following:

**Declarative model-to-model and model-to-text transformations.** Model transformations provide automation for bridges between artifacts of an MDE workflow; see Figure 1.1 for roles of MT. Transformations are commonly defined in rule-based declarative formalisms. Queries are used for specifying when and where one can apply the rules of the transformation specification; query evaluation then involves processing the source model to find the parts that will be transformed into the target model according to the rule.

**Simulation of behavioral models with operational semantics defined using rules.** Model simulation is the representation of system states and the application of state changes (transitions, evolution paths) to reach different system states, as described by the behavioral model. Model simulation is used for various dynamic analysis techniques such as model checking \cite{JRG12}, design space exploration \cite{HV10}, or stochastic simulation of trajectories to characterize typical behavior \cite{3}. These analysis techniques may be used to verify a system under design, to assess its properties, and/or to support designing a safe and efficient system.
Once again, the precondition of a transition rule is essentially a query that finds the applicable transitions in any given state of the model.

**Analysis and reporting on models.** Many kinds of static analysis can be formulated declaratively in a model query formalism, including gathering aggregated statistics, discovering correspondences of elements, or design rule checks by finding violations of well-formedness constraints and modeling conventions. In static analysis and reporting, queries can be used to provide a very direct and immediate feedback to the engineers.

### 1.2.2 Challenges

As models in engineering practice are often subject to change, their evolving nature raises many challenges with respect to the engineering process and model transformation in particular. These problems may range from organizational issues of a change request approval process to propagating or migrating changes between models (see also Section 1.1.3). My thesis focuses on a single overall challenge: **model queries over evolving models**.

Solving this challenge may present various ways to improve the engineering process in the previously indicated use cases. Source incrementality may radically increase query performance in model-to-model transformation, simulation and static analysis. In some cases, this might bring a qualitative improvement in addition to the quantitative one. For instance, immediate feedback in static verification may elevate modeling to a highly productive interactive process. Finally, extending the query language into a change-driven formalism may make the specification of change-propagating transformations and evolutionary analysis easier.

This top-level challenge naturally involves the following aspects:

**Language.** The first challenge is finding a query language that is:

- expressive enough to capture complex relationships of model elements such as rich structural interconnections, attribute conditions, quantification, aggregation and transitive reachability;
- concise enough to formulate complex relationships in a straightforward way, without wasting effort;
- compositional to support top-down or bottom-up thinking and reuse;
- intuitive to understand in light of its direct correspondence to model structure;
- able to express conditions relating to the change between two versions of a model, in addition to the structure of a single static model, in order to support change-driven transformation specification;
- declarative in order to support various evaluation strategies.

**Evaluation method.** The second challenge is finding an evaluation strategy to the declarative query language that is:

- source-incremental;
- efficient in terms of execution time, and also regarding memory footprint;
- capable of taking advantage of the parallel execution provided by modern symmetric multiprocessing hardware;
• supports the features of the query language.

**Adaptation to technological platforms.** The final challenge is realizing both the language concepts and the evaluation engine in context of technologies with industrial relevance, namely:

• primarily the Eclipse Modeling Framework (EMF);
• alternatively relational databases, especially in-memory implementations;
• while addressing compatibility with all scenarios of model evolution in case of change-driven execution.

### 1.2.3 Contributions of the thesis

My thesis will offer the following improvements over the state-of-the-art of model query technology:

• I propose an incremental evaluation strategy for model queries formulated as graph patterns, and demonstrate its efficiency. (Contribution 1)

• I integrate this strategy into the industrial Eclipse Modeling Framework by designing a run-time translation layer and adapting the query syntax, and evaluate the performance of the resulting solution. (Contribution 2)

• I extend the query formalism to support change-driven transformations by transparently capturing changes of the model regardless of the scenario, and design scenario-specific strategies for execution. (Contribution 3)

• I provide bidirectional model synchronization, change impact analysis and consistency checking in the domain of security requirement engineering, by applying the above techniques. (Contribution 4)

For each of these contributions, Figure 1.2 depicts the challenges that are addressed, as well as the application domains and use cases where the results are demonstrated.

### 1.3 The structure of the thesis

• First Chapter 2 will provide the background knowledge that is necessary to follow the new scientific results of later chapters, as well as introduce the terminology I will use throughout the thesis.

• The following chapters address Contribution 1.

  • Chapter 3 presents the first of the new contributions: a formal treatment of incremental pattern matching, the adaptation of an incremental evaluation algorithm to the language of graph patterns and the previously introduced formalization, and finally empirical evaluation.

  • Chapter 4 follows by introducing a number of extensions regarding language, execution strategy and platform technology.

• Chapter 5 presents the realization of this solution on the modeling platform of EMF, including questions of language design, overcoming technological hurdles, and again experimental validation of performance, thereby fulfilling Contribution 2.
• Chapter 6 deals with Contribution 4, demonstrating the application of thesis results in a domain-specific modeling environment from the domain of security requirement engineering. After presenting the tool environment, queries and transformations will be shown validating security properties, analyzing change impact on human arguments, and providing live synchronization between model representations. Some of these problems are beyond the previously proposed techniques, providing a source of motivation for Chapter 7.

• Chapter 7 introduces change-driven transformations, a novel framework for specifying reactive behavior for evolving models, according to Contribution 3. After clarifying terminology and analyzing possible application scenarios, language and semantics will be provided for change-driven transformations, as well as execution algorithms. The new techniques will then be demonstrated on case studies previously introduced in Chapter 6, followed by a detailed evaluation and discussion of the new approach.

• Finally, Chapter 8 concludes the thesis by summarizing the new scientific results that were achieved by fulfilling the goals of Section 1.2.3, as well as their application and dissemination.
Chapter

2

Background

This chapter will lay down the formal foundations of my thesis, upon which novel scientific contributions will be built in subsequent chapters. Section 2.1 will present a formalization of models, metamodels, query and manipulation operations, which are central concepts of MDE. As a declarative formalism for specifying query and manipulation operations, graph patterns and graph transformation will be introduced in Section 2.2.

2.1 Modeling preliminaries

This section introduces the basic modeling concepts that are a prerequisite to understanding the pattern matching process. Table 2.1 shows an organized summary of the names, notations and relationships of some modeling concepts that will be discussed here, as well as the notions of pattern variables and constraints that will be introduced later in Section 2.2.

2.1.1 Running example: Petri nets

In the current thesis, the modeling language of Petri nets will be used as a DSM example, while the dynamic simulation of Petri nets will serve as a performance benchmark. Petri nets (such as the sample in Figure 2.1(a)) are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. Yet they are simple enough to briefly demonstrate the main concepts used in this chapter.

In the core formalism used throughout the thesis, Petri nets are bipartite graphs, with two disjoint sets of nodes: Places and Transitions. Places may contain an arbitrary non-negative number

<table>
<thead>
<tr>
<th>Types</th>
<th>Model elements</th>
<th>Sources</th>
<th>Targets</th>
<th>Pattern elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Cls_{Str}$</td>
<td>$Ent_{Str}$</td>
<td>$Ent_{Dat}$</td>
<td>N/A</td>
<td>$V_{ent}$</td>
</tr>
<tr>
<td>$Cls_{Dat}$</td>
<td></td>
<td></td>
<td>N/A</td>
<td>$C_{ent}$</td>
</tr>
<tr>
<td>$Fea_{Str}$</td>
<td>$Rel_{Str}$</td>
<td>$Ent_{Str}$</td>
<td>$Ent_{Dat}$</td>
<td>$V_{rel}$</td>
</tr>
<tr>
<td>$Fea_{Dat}$</td>
<td></td>
<td>$Ent_{Dat}$</td>
<td></td>
<td>$C_{rel}$</td>
</tr>
<tr>
<td>$Fea_{Val}$</td>
<td>$Rel_{Val}$</td>
<td>$Ent_{Str}$</td>
<td>$Ent_{Dat}$</td>
<td>$V_{rel}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$Ent_{Dat}$</td>
<td></td>
<td>$C_{rel}$</td>
</tr>
</tbody>
</table>

Table 2.1: Notational summary of modeling concepts
of (indistinguishable) Tokens. Such a token distribution (marking) defines the state of the modeled system.

The state of the net can be changed by firing enabled transitions. A transition is enabled (fireable) if each of its input places (connected to the transition via input arcs) contains at least one token, and furthermore no place connected via an inhibitor arc contains any tokens. When firing a transition, we remove a token from all input places and add a token to all output places (as defined by output arcs).

Figure 2.1(a) shows a sample Petri-net in its domain-specific concrete syntax. Each hollow circle represents a place, while each elongated black block represents a transition. Black circles within a place indicate tokens belonging to that place. Input arcs are depicted as arrows pointing from a place to a transition, output arcs are arrows pointing from a transition to a place. Inhibitor arcs look similar to input arcs, but terminate in a small circular disc instead of an arrowhead.

**Example 1** In the example Petri-net of Figure 2.1(a), both transitions \( t_1 \) and \( t_2 \) are fireable. If one fires \( t_1 \) once (see Figure 2.1(b)), it will reduce the token count of \( p_1 \) to 2, while \( p_3 \) will continue to have a single token; both transitions will remain fireable. Firing \( t_2 \) (see Figure 2.1(c)), on the other hand, would remove a token from \( p_3 \) and add one to \( p_2 \); after this step neither transitions are enabled anymore.

### 2.1.2 Graph models and metamodeling

The following paragraphs introduce a formal foundation of graph models. Note that several approaches in literature are largely similar, such as graph schemata [SWZ99], type graphs with inheritance [TR05], certain semantics of MOF [AP08], or VPM [Var04].

**Definition 1 (Universe)** The universe \( U \) is an infinite set consisting of all potential model and metamodel elements, as well as all potential data values (numbers, text strings, etc.).
Definition 2 (Metamodel) A metamodel is a structure $MM = \langle Cls, Fea, owner, range, super \rangle$ where $Cls \subseteq U$ is a set of classifiers (node / entity types), $Fea \subseteq U$ is a set of features (edge / relation types) disjoint from $Cls$; $owner, range : Fea \rightarrow Cls$ maps the features to their owner (source) classifiers and range (target) types, respectively. The reflexive partial order $super = \subseteq (Elements_{MM} \times Elements_{MM})$ defines supertyping as a binary relation, where $Elements_{MM} = Cls \cup Fea \subseteq U$ is the set of metamodel elements, and $super = super^{cls} \cup super^{fea}$ is composed of reflexive partial orders $super^{cls} \subseteq (Cls \times Cls)$ and $super^{fea} \subseteq (Fea \times Fea)$, with $\cup$ denoting a disjoint union.

If $super(C, D)$, we say that $D$ is a supertype of $C$ and $C$ is a subtype of $D$ (permitting equality). Note that the supertyping relationship is defined here as a partial order (reflexive, antisymmetric, transitive). In some implementations and alternate formalizations, only some supertype assertions are explicit, and the whole partial order is induced transitively.

Definition 3 (Well-formed metamodel) A metamodel is well-formed iff the $owner$ and $range$ maps are homomorphic w.r.t supertyping, i.e. $\forall (f_1, f_2) \in super^{fea} : \langle owner(f_1), owner(f_2) \rangle \in super \land \langle range(f_1), range(f_2) \rangle \in super$.

The set of all well-formed metamodels is denoted by $Meta$. From now on, we consider such metamodels only.

Example 2 A metamodel of the domain of Petri-nets could consist of the classifiers Place, Token and Transition, and features\(^1\) OutArc (owner: Transition: range: Place), InArc (owner: Place: range: Transition), InhibitorArc (owner: Place: range: Transition) and Marking (owner: Place: range: Token).

To demonstrate supertyping and support the discussion of Petri-net model elements in general, a classifier PetriNode can be introduced as a common supertype of Place and Transition (types visually indicated as nodes), while PetriEntity can be a supertype of Token in addition to PetriNode and its subtypes. The feature PetriEdge (owner: range: PetriNode), representing all visual edges, is a common supertype of OutArc, InArc and InhibitorArc. It is easy to see that the metamodel is well-formed, even after the introduction of these feature supertyping relationships; e.g. the owner and range of OutArc is Transition and Place respectively, both of which are subtypes of PetriNode, which is the owner and range of feature supertype PetriEdge. Altogether, the entire supertyping partial order would be $super = \{ \langle PetriEntity, PetriEntity \rangle, \langle Token, Token \rangle, \langle Token, PetriEntity \rangle, \langle PetriNode, PetriNode \rangle, \langle PetriNode, PetriEntity \rangle, \langle Place, Place \rangle, \langle Place, PetriNode \rangle, \langle Place, PetriEntity \rangle, \langle Transition, Transition \rangle, \langle Transition, PetriNode \rangle, \langle Transition, PetriEntity \rangle, \langle Marking, Marking \rangle, \langle PetriEdge, PetriEdge \rangle, \langle InArc, InArc \rangle, \langle InArc, PetriEdge \rangle, \langle OutArc, OutArc \rangle, \langle OutArc, PetriEdge \rangle, \langle InhibitorArc, PetriNode \rangle, \langle InhibitorArc, PetriEntity \rangle \}$.\(^2\)

This example metamodel is visually depicted in Figure 2.2. Rectangles represent classifiers, and thicker arrows represent features (pointing from the owner to the range). Dashed lines ending in hollow triangles are explicit supertype assertions (pointing from the subtype to the supertype); the actual $super$ relationship would be the reflexive transitive closure of these explicit edges.

Definition 4 (Graph model) A graph model conforming to a well-formed metamodel $MM = \langle Cls, Fea, owner, range, super \rangle \in Meta$ is a structure $G = \langle Ent, Rel, src, trg, typ \rangle$ where $Ent \subseteq U$ is a set of entities (graph vertices / nodes), $Rel \subseteq U$ is a set of relations (graph arcs / edges); $src, trg : Rel \rightarrow Ent$ maps the relations to their source and target entities, respectively; the

---

\(^1\)According to some conventions, features should be spelt with lowercase initials; this convention is not followed here.
typing of elements is the function \( \text{typ} : \text{Elements}_G \to \text{Elements}_MM \) where \( \text{Elements}_G \subset U \) is an abbreviation for the set of graph elements \( \text{Ent} \cup \text{Rel} \); and finally the following properties are met by the typing function \( \text{typ} \):

- \( \forall e \in \text{Ent} : \text{typ}(e) \in \text{Cls} \)
- \( \forall r \in \text{Rel} : \text{typ}(r) \in \text{Fea} \)
- \( \forall r \in \text{Rel} : \text{super}(\text{typ(src(r))}, \text{owner}(\text{typ}(r))) \land \text{super}(\text{typ(trg(r))}, \text{range}(\text{typ}(r))) \)

The concept of instantiation will be useful for discussing graph models both on the model level and on the level of individual model elements.

**Definition 5 (Instantiation (model level))** Graph model \( G \) instantiates (alternatively “is an instance of” or “is defined by”) metamodel \( MM \), denoted as \( G : MM \), if and only if \( G \) is conforming to \( MM \). The set of all graph models instantiating a given metamodel \( MM \in \text{Meta} \) will be denoted as \( \text{Graphs}_{MM} = \{ G \mid G : MM \} \).

**Definition 6 (Instantiation (element level))** In a graph model \( G = \langle \text{Ent}, \text{Rel}, \text{src}, \text{trg}, \text{typ} \rangle \) conforming to a well-formed metamodel \( MM = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle \), the graph element \( e \in \text{Elements}_G \) instantiates (or is an instance of) type \( t \in \text{Elements}_MM \) iff \( \text{super(\text{typ}(e)), \text{owner}(\text{typ}(e))) \land \text{super(\text{typ}(\text{trg}(r)), \text{range}(\text{typ}(r)))} \)

Using the terminology of instantiation, we can restate the third property of the typing function in Definition 4: the source and target of a relation must instantiate the owner type and the range type of the relation type, respectively. Due to the well-formedness of the metamodel, this can be further rephrased in the following way to take feature supertypes into account: the source/target of an instance of a feature must instantiate the owner/range of the feature.

It is also possible to use a similar schema to represent hypergraphs, where relations / hyperedges may have more than two incidence maps instead of just \( src \) and \( trg \). Metamodels in such formalisms employ more edge type maps in addition to just \( \text{owner} \) and \( \text{range} \). While detailed definitions are
omitted for conciseness, all results in the thesis can be generalized very easily to hypergraphs, and the reader is assumed to be familiar with the concept.

In engineering practice, metamodels are usually represented in an extended formalism containing helpful practical information, such as names of classifiers and features, or additional conditions of conformance (e.g. multiplicity constraints in the metamodel may impose a limit on the number of relations of a given type that may be incident on a single entity). These additional metamodel elements are not formalized here, as they have limited impact on the topics discussed in the thesis.

In some formalisms with multi-level metamodeling, the metamodel itself is represented as an instance model of a meta-metamodel, which is also an instance model of a meta-meta-model, etc. In fact, all of these meta-levels can be considered part of the same graph model. Supporting such systems requires careful semantic considerations, such as regarding what happens when the type of an element is modified. For the sake of simplicity, multi-level metamodeling is considered out of scope for the thesis.

Example 3 Revisiting the sample Petri-net in Figure 2.1(a), it can now be interpreted as a graph model conforming to the metamodel of Example 2 (see Figure 2.2). The abstract syntax (logical structure) of the graph model is shown in Figure 2.3, with hexagons depicting entities and arrows depicting relations, both organized vertically by type. Let us contrast with the same model in the concrete syntax notation specific to Petri-nets (Figure 2.1(a)). The entities $p_1$, $p_2$ and $p_3$ have the type Place (thereby instantiating Place, PetriNode and PetriEntity), $t_1$, $t_2$ have the type Transition, and $k_1$, $k_2$, $k_3$, $k_4$ (not labeled in Figure 2.1(a)) are of type Token. Relations of type InArc are $i_1$ from source $p_1$ to target $t_1$, $i_2$ from $p_3$ to $t_1$ and $i_3$ from $p_3$ to $t_2$. The OutArc instances are $o_1$ from $t_1$ to $p_3$ and $o_2$ from $t_2$ to $p_2$. The only InhibitorArc is $g_1$ from $p_2$ to $t_1$. Finally, there are the Marking relations (not depicted in Figure 2.1(a)); from the source $p_1$ there are $m_1$, $m_2$, $m_3$ with targets $k_1$, $k_2$, $k_3$, respectively; while $m_4$ goes from $p_3$ to $k_4$.

2.1.3 Operations on metamodels

The goal of this section is to introduce different kinds of composition of metamodels, that will be useful in describing attributed graph models. The chosen approach is similar but not identical to
UML package merge (formalized in [ZD06]), as it provides formal treatment of (a) supertyping, (b) relation types forming an unidirectional glue between two metamodels, and (c) the decomposition of instance models along the structure of the metamodel. More metamodel operations are available in [ES06].

For the purposes of this thesis, it is enough to analyze metamodel merging in three special cases. This way, it will be trivial to preserve the partial order property of supertyping and the well-formedness property of metamodels.

**Definition 7 (Disjoint metamodels)** Metamodels $MM_1 = \langle \text{Cls}_1, \text{Fea}_1, \text{owner}_1, \text{range}_1, \text{super}_1 \rangle$ and $MM_2 = \langle \text{Cls}_2, \text{Fea}_2, \text{owner}_2, \text{range}_2, \text{super}_2 \rangle$ are feature-disjoint iff $\text{Fea}_1 \cap \text{Fea}_2 = \emptyset$ and totally disjoint iff $\text{Cls}_1 \cap \text{Cls}_2 = \emptyset$ and $\text{Fea}_1 \cap \text{Fea}_2 = \emptyset$.

**Definition 8 (Disjoint merge of metamodels)** For the totally disjoint metamodels $MM_1 = \langle \text{Cls}_1, \text{Fea}_1, \text{owner}_1, \text{range}_1, \text{super}_1 \rangle$ and $MM_2 = \langle \text{Cls}_2, \text{Fea}_2, \text{owner}_2, \text{range}_2, \text{super}_2 \rangle$, their disjoint merge is a metamodel $MM_1 \cup MM_2 = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle$ with $\text{Cls} = \text{Cls}_1 \cup \text{Cls}_2$, $\text{Fea} = \text{Fea}_1 \cup \text{Fea}_2$, $\text{owner} = \text{owner}_1 \cup \text{owner}_2$, $\text{range} = \text{range}_1 \cup \text{range}_2$, $\text{super} = \text{super}_1 \cup \text{super}_2$.

The supertyping relationship $\text{super}$ remains a partial order because $\text{super}_1$ and $\text{super}_2$ are partial orders on disjoint sets. Trivially, if the constituent metamodels $MM_1$ and $MM_2$ are well-formed, so is the resulting $MM$.

This first merge operation deals with two metamodels describing two completely independent domains. In this case it is trivial that we can obtain a merged metamodel whose instance models may contain elements of either kind. For example, if we take the Petri-net metamodel $MM_{\text{Petri}}$ of Example 2 and the totally disjoint metamodel $MM_{\text{Process}}$ of process flows, instance models of $MM_1 \cup MM_2$ may contain Petri-nets and processes, but no connections between them.

**Definition 9 (Feature-merge of metamodels)** For the pair of feature-disjoint metamodels $MM_1$ and $MM_2$ sharing their classifiers $\text{Cls}$ along with classifier supertyping $\text{super}^{\text{cls}}$, where $MM_1 = \langle \text{Cls}, \text{Fea}_1, \text{owner}_1, \text{range}_1, \text{super}^{\text{cls}}_1 \cup \text{super}^{\text{fe}a}_1 \rangle$ and $MM_2 = \langle \text{Cls}, \text{Fea}_2, \text{owner}_2, \text{range}_2, \text{super}^{\text{cls}}_2 \cup \text{super}^{\text{fe}a}_2 \rangle$, their feature-merge is a metamodel $MM_1 \uplus MM_2 = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle$ with $\text{Fea} = \text{Fea}_1 \cup \text{Fea}_2$, $\text{owner} = \text{owner}_1 \cup \text{owner}_2$, $\text{range} = \text{range}_1 \cup \text{range}_2$, $\text{super} = \text{super}^{\text{cls}}_1 \cup \text{super}^{\text{fe}a}_1 \cup \text{super}^{\text{cls}}_2 \cup \text{super}^{\text{fe}a}_2$.

Now the supertyping relationship $\text{super}$, while preserving all supertyping information from $MM_1$ and $MM_2$, remains a partial order because $\text{super}^{\text{cls}}$ and $\text{super}^{\text{fe}a}_1$ and $\text{super}^{\text{fe}a}_2$ are partial orders on pairwise disjoint sets. Well-formedness is preserved in the resulting metamodel $MM$ for the trivial reasons.

This second operation deals with the composition of two disjoint set of features over the same set of classifiers; models instantiating the merged metamodel may contain entities that instantiate the common classifiers, and relations typed by either set of features. For example, the $MM_{\text{Petri}}$ of Example 2 (without the relation supertype PetriEdge) could be decomposed as $MM_{\text{Petri}} = MM_{\text{Petri,Arches}} \uplus MM_{\text{Petri,Marking}}$, where $MM_{\text{Petri,Arches}}$ only includes the arc features, the feature set of $MM_{\text{Petri,Marking}}$ is reduced to $\{\text{Marking}\}$, but both have the original set of classifiers.

**Definition 10 (Unidirectional glue between metamodels)** For totally disjoint metamodels $MM_1 = \langle \text{Cls}_1, \text{Fea}_1, \text{owner}_1, \text{range}_1, \text{super}_1 \rangle$ and $MM_2 = \langle \text{Cls}_2, \text{Fea}_2, \text{owner}_2, \text{range}_2, \text{super}_2 \rangle$, the metamodel $MM = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle$ is
2.1. MODELING PRELIMINARIES

Figure 2.4: Petri-net metamodel glue-merged with a process metamodel

a unidirectional glue from $MM_1$ to $MM_2$ iff $MM$ is feature-disjoint from $MM_1$ and $MM_2$, with $Cls = Cls_1 \cup Cls_2$, $super^{cls} = super^{cls}_1 \cup super^{cls}_2$ and finally $owner(Fea) \subseteq Cls_1 \land range(Fea) \subseteq Cls_2$.

In other words, the glue metamodel only contains features pointing from the first metamodel $MM_1$ to the second metamodel $MM_2$. It would be equally easy to define the concept of bidirectional glue between metamodels.

Definition 11 (Glue-merge of metamodels) For the totally disjoint metamodels $MM_1 = \langle Cls_1, Fea_1, owner_1, range_1, super_1 \rangle$ and $MM_2 = \langle Cls_2, Fea_2, owner_2, range_2, super_2 \rangle$ and unidirectional glue $MM_{glue} = \langle Cls_1 \cup Cls_2, Fea_{glue}, owner_{glue}, range_{glue}, super_{glue} \rangle$ from $MM_1$ to $MM_2$, their glue-merge is the metamodel $MM_1 \xrightarrow{MM_{glue}} MM_2 = (MM_1 \cup MM_2) \cup MM_{glue}$.

The feature-merge operation is applicable here, since the glue and the result of the disjoint merge share their classifiers and are feature-disjoint. If all three initial metamodels are well-formed, so is the result of the glue-merge, due to the preserving properties of the two previous merge operations. Note that in general, disjoint merges are special cases of the glue-merge.

This third kind of metamodel operation lets us compose complex metamodels from simpler ones and their interconnections. As a quick example, let us assume that Petri-nets are automatically generated from process models, and we would like to preserve traceability from Petri-net elements to the corresponding process model elements. It is possible to define an unidirectional glue metamodel $MM_{glue}$ between $MM_{Petri}$ and $MM_{Process}$ that would contain all Petri-net and process classifiers (along with their subtyping relationships), and various traceability features pointing from a Petri net element to a process element (such as $trFork$ pointing from a $Transition$ element in the Petri-net to the $Fork$ node in the process model that the transition was generated from). Model instantiating the glue-merge $MM_{Petri} \xrightarrow{MM_{glue}} MM_{Process}$ (illustrated on Figure 2.4) therefore contain Petri-nets, process models, and relations instantiating the traceability features, pointing from the Petri-net elements to the process elements.

Definition 12 (Decomposition of instance models along a glue-merge) A graph model $G = \langle Ent, Rel, src, trg, typ \rangle$ instantiating a glued metamodel $MM_1 \xrightarrow{MM_{glue}} MM_2$ decomposes along the glue-merge into $\langle G_1, Rel_{glue}, G_2 \rangle$, where $i \in \{1, 2\} : G_i = \langle Ent_i, Rel_i, src_i, trg_i, typ_i \rangle$ with $Ent_i = \{ r \mid r \in Ent \land typ(r) \in Cls_i \}$, $Rel_i = \{ r \mid r \in Rel \land typ(r) \in Fea_i \}$; $src_i, trg_i$ and $typ_i$ are restricted on the corresponding sets; and finally $Rel_{glue} = \{ r \mid r \in Rel \land typ(r) \in Fea_{glue} \}$.

In essence, an instance model of a glue-merged metamodel can be decomposed into instances of the two glued metamodels, and glue relations connecting the first one to the second. Continuing
the previous example, instances of $MM_{Petri} \xrightarrow{MM_{true}} MM_{Process}$ would decompose into a simple Petri-net model, a simple process model, and a set of Petri-net-to-process traceability relations.

2.1.4 Attribute values

While the previous definition of graph model is sufficient for describing abstract structures, most practical purposes also require the assignment of textual, numerical, categorical or other sorts of attributes. This assumes that there are structural entities determining the structure of the model, as well as a pool of potential attribute values that can be assigned to structural entities. Relationships between these attribute values (e.g. ordering or operators) are also modeled. See [Kas06] for a more formal treatment of data algebrae in graph modeling, which heavily influenced the following definitions, or [EEPT06] for a category-theory-based formulation.

Definition 13 (Data values and data algebra) $Ent_{Dat} \subseteq U$ is the immutable and infinite set of all potential attribute values. These values are classified into special attribute types (such as integers, strings, enumerable categories, etc.), also called datatypes, denoted as $Cls_{Dat} \subseteq U$. The data algebra $Dat$ is the graph model $\langle Ent_{Dat}, Rel_{Dat}, src_{Dat}, trg_{Dat}, typ_{Dat} \rangle$, which also introduces data relations $Rel_{Dat} \subseteq U$, the immutable and infinite set of relations between data values (e.g. ordering, substring, etc.), typed by so-called data predicates $Fea_{Dat} \subseteq U$. $Dat$ instantiates the well-formed metamodel called data signature $MM_{Dat} = \langle Cls_{Dat}, Fea_{Dat}, owner_{Dat}, range_{Dat}, super_{Dat} \rangle$.

Relations of higher arity (e.g. operators such as multiplication, concatenation, etc.) can be similarly represented using hypergraphs, or by auxiliary nodes. For a simple example involving auxiliary nodes, let us assume the data signature $MM_{Dat}$ contains a data type and an auxiliary node type $Number, Division \in Cls_{Dat}$, and data predicates $dividend, divisor, quotient \in Fea_{Dat}$ with $Division$ as owner and $Number$ as range. Instantiating this data signature, the fact that $3/4 = 0.75$ can be represented by the auxiliary node $\langle 3, 4, 0.75 \rangle : Division \in Ent_{Dat}$, with three outgoing relations (elements in $Rel_{Dat}$) pointing to numbers: one of type $dividend$ to 3, one of type $divisor$ to 4, and one of type $quotient$ to 0.75. Neither the option of auxiliary nodes nor the alternative hypergraph-based formalization is explored in detail here, but it is assumed that higher-arity relations are available.

Definition 14 (Metamodel of attributed graph) An attributed graph metamodel over data $Dat$ is the glue-merge $MM_{Str} \xrightarrow{MM_{Val}} MM_{Dat} = MM = \langle Cls, Fea, owner, range, super \rangle$ for some structural metamodel $MM_{Str} = \langle Cls_{Str}, Fea_{Str}, owner_{Str}, range_{Str}, super_{Str} \rangle$ that is totally disjoint from $MM_{Dat}$, and for appropriate $Fea_{Val}$ glue features. $Cls_{Str}$ is the set of classes (structural classifiers), $Fea_{Str}$ are the so-called structural associations. $Fea_{Val}$ are called attribute names with $owner(Fea_{Val}) \subseteq Cls_{Str}$, $\wedge range(Fea_{Val}) \subseteq Cls_{Dat}$ (i.e. from classes to datatypes).

Definition 15 (Attributed graph model) An attributed graph is any graph model $G = \langle Ent, Rel, src, trg, typ \rangle$ conforming to an attributed metamodel $MM_{Str} \xrightarrow{MM_{Val}} MM_{Dat} = MM = \langle Cls, Fea, owner, range, super \rangle$ that decomposes along the glue-merge into $(G_{Str}, Rel_{Val}, G_{Dat})$, where $G_{Dat} = Dat$ (i.e. $G$ must contain $Dat$ as its sub-model). $Ent_{Str}$ is called the set of structural entities, $Rel_{Str}$ is the set of structural relations, and $Rel_{Val}$ is the set of value assignment relations that assign attribute values to structural entities. $G \models obj.attr = val$ denotes the fact that $\exists r \in Rel_{Val}(G)$ where $src(r) = obj, trg(r) = val$ and $typ(r) = attr$. 
In most practical cases, the structural instance model \( G_{Str} \) and value assignments \( Rel_{Val} \) are required to be finite. Another frequent restriction is that each structural entity must have at most one outgoing relation per attribute name (many-to-one multiplicity, may be indicated in the metamodel as well).

Obviously, any implementation in a finite computer can only manifest a finite subset of the infinite \( Dat \) at a time, such as \( \text{trg}(Rel_{Val}) \). Because of the infinite degrees, \( Rel_{Dat} \) arcs are rarely stored or enumerated even for those parts of \( Ent_{Dat} \) that are kept in memory; but it is assumed that the existence of a data relation of a given type between given data entities is easily decidable. Some predicate types in \( Fea_{Dat} \) might be function-like, meaning that one (or more) of their incident nodes can be efficiently deduced from the others (e.g., the value of a product is derivable from the value of its factors, or the value of a complex arithmetic expression is computable from the value of its free variables).

**Example 4**  
An alternate Petri-net metamodel (see Figure 2.5) serves as an example application of the attributed graph concept. The key idea is that tokens are indistinguishable, therefore it is enough to keep track of the number of tokens at a given place. Most of the previous metamodel is retained as the structural metamodel (depicted as solid boxes and arrows). Without a Token class, the Marking feature is now an attribute name (depicted as a dashed arrow) pointing to the Integer datatype (depicted as a dashed box). There is also a data predicate Successor (depicted as striped arrow) with Integer as owner and range; its instances are pointing from each number to the subsequent one. Note that the supertypes were omitted for the sake of simplicity.

The instance model depicted in Figure 2.1(a) can be described by a different graph model conforming to the new, attributed metamodel; the new abstract syntax is depicted in Figure 2.6. First of all, the infinite data algebra part of the graph model consist of all integer numbers, each connected to the next by a data relation \((s_0, s_1, s_2, \ldots)\) of type Successor. The structural graph consist of the already discussed instances (see Example 3) of Place, Transition and the three Arc associations. The two are glued together by value assignment relations of type Marking: \( m_1 \) has \( p_1 \) as source and the data entity 3 as target; \( m_2 \) points from \( p_2 \) to 0, and finally \( m_3 \) goes from \( p_3 \) to 1.

### 2.1.5 Model access operations

In addition to the static structure of graph models, it is also important to study their evolution over time, and identify the various operations that access the model. Two main kinds of operations can be distinguished: *queries* that do not change the model but yield useful output, and *manipulation* operations that change the model.
Definition 16 (Model access operation) The model access operation \( Op : \text{Graphs}_{MM} \rightarrow (\text{Graphs}_{MM} \times \text{Out}_{Op}) \) over metamodel \( MM = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle \) is a partial function that maps a graph model \( G \in \text{Graphs}_{MM} \) to the tuple \( (G', \text{out}) \) with result model \( G' \in \text{Graphs}_{MM} \) and output \( \text{out} \in \text{Out}_{Op} \) from output range \( \text{Out}_{Op} \subseteq 2^{D_G \times D_G \times ... \times D_G} \) where \( D_G = \text{Elements}_G \cup \text{Elements}_{MM} \). The application of the operation on graph model \( G \) is denoted as \( G.Op \), while \( G.Op.r \) denotes the result model and \( G.Op.out \) denotes the yielded output so that \( G.Op = (G.Op.r, G.Op.out) \). The set \( \text{Dom}(Op) \) of models on which \( Op \) can be applied is indicated by the precondition of \( Op \).

In essence, if the precondition permits, a model access operation may transform an actual graph model to an updated graph model, and yield an output that is a set of tuples formed of graph elements and metamodel elements.

2.1.5.1 Query operations

Definition 17 (Graph query operation) The graph query operation \( Q \) over metamodel \( MM = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle \) is a model access operation with \( G.Q.r \equiv G \) for \( \forall G \in \text{Dom}(Q) \).

Note that the above formalization imposes no restrictions on the specification and internal structure of query operations. However, there is a fixed set of elementary model query operations on graph model \( G = \langle \text{Ent}, \text{Rel}, \text{src}, \text{trg}, \text{typ} \rangle : MM = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle \). These queries are organized into four families of parameterized operations: entity queries \( \text{Query}_1(E : C) \) and \( \text{Query}_2(E :: C) \) are query operations for each \( E, C \in \text{Universe} \setminus \{\ast\} \) (where \( \ast \not\in \text{Universe} \) is a special token external to the universe); as well as relation queries \( \text{Query}_3(E_s(R:F) \rightarrow E_t) \) and \( \text{Query}_4(E_s(R::F) \rightarrow E_t) \) for each \( R, F, E_s, E_t \in \text{Universe} \setminus \{\ast\} \). Here \( E : C \) and \( E :: C \) are used as notational shorthands for the tuple \( (E, C) \) while \( E_s(R:F) \rightarrow E_t \) and \( E_s(R::F) \rightarrow E_t \) are a shorthand for tuple \( (R, F, E_s, E_t) \). The subscripts \( \text{Query}_i \) can be inferred from the notation used to parameterize the operation, so they will be omitted for brevity.

These elementary query operations have no precondition \( (\text{Dom}(Q) = \text{Graphs}_{MM}) \), and are specified in the following paragraphs by the output yielded by their application on a model:
G.Query\((E : C)\) queries the graph model \(G\) for an entity \(E\) as an instance of classifier \(C\). Both parameters \(E\) and \(C\) can be specified as a concrete entity respectively classifier to restrict the results of the query, or can be given as the special token \(*\) that allows all values of the parameter. The result of the query is all valid entity-classifier pairs (restricted by the input parameters): 
\[
G.\text{Query}(E : C).out \; := \; \{ \langle e, c \rangle \mid e \in \text{Ent} \land G \models e : c \land (E \neq * \implies e = E) \land (C \neq * \implies c = C) \}.
\]

G.Query\((E :: C)\) queries the graph model \(G\) for an entity \(E\) with the classifier \(C\) as its direct type: 
\[
G.\text{Query}(E :: C).out \; := \; \{ \langle e, c \rangle \mid (e, c) \in G.\text{Query}(E : C).out \land \text{typ}(e) = c \}. \text{ The } :: \text{ symbol distinguishes this query from the previous one (with the } : \text{ symbol) which includes supertypes in addition to the direct type.}
\]

G.Query\((E_s (R:F) E_t)\) queries the graph model \(G\) for the relation \(R\) as an instance of feature \(F\), pointing from entity \(E_s\) to \(E_t\). All four parameters can be specified as a concrete element to restrict the query, or can be given as the special token \(*\) that allows all values of the parameter. The result of the query is all valid relation-classifier-source-target tuples (restricted by the input parameters): 
\[
G.\text{Query}(E_s (R:F) E_t).out \; := \; \{ \langle r, f, e_s, e_t \rangle \mid r \in \text{Rel} \land G \models r : f \land \text{src}(r) = e_s \land \text{trg}(r) = e_t \land (R \neq * \implies r = R) \land (F \neq * \implies f = F) \land (E_s \neq * \implies e_s = E_s) \land (E_t \neq * \implies e_t = E_t) \}.
\]

In attributed graphs, depending on details of data algebra \(Dat\), there may also be further limitations on queries regarding instances of \(Cls_{Dat}\) and \(Fea_{Dat}\) due to practical difficulties in enumerating infinite sets. Typically, instances of a datatype are not possible to enumerate; instances of some data predicates, however, are finitely enumerable if the source or target entity is specified as a concrete value. For instance, one can find out the upper or lower neighbor of a given integer by a model query of type \textit{Successor} (see Example 4). In a more general case, even if a source does not correspond to a single target, the set of targets may be enumerable by a finite computation.

\[\textbf{Definition 18 (Enumerable and functional data predicates)}\] A data predicate \(f \in Fea_{Dat}\) is \textit{enumerable by source} if the value \(e_s\) of the source data entity corresponds to a set of valid values for the data relation \(r\) that is enumerable using finite resources, i.e. \(f(e_s) = Dat.\text{Query}(e_s \xleftarrow{s} r \xrightarrow{} *).\text{out}\) is finitely computable. Analogously, a data predicate is \textit{enumerable by target} if valid instantiating data relations can be enumerated for any given value of the target data entity.

Enumerable data predicates like this will play an important role later in attributed graph pattern matching. The above definition can be generalized for hypergraph data algebrae, e.g. a \textit{Division} data predicate is functionally determined by any two of its three operands \textit{dividend}, \textit{divisor}, \textit{quotient}. Note that data predicates are always assumed to be functionally determined by the full set of their entities (e.g. a source and the target will always uniquely identify a data relation of a given binary data predicate).

As a final observation, any graph model \(G\) can be completely and uniquely reconstructed from the joint query results \(G.\text{Query}(* :: *).out\) and \(G.\text{Query}(* \xrightarrow{::} *).out\), i.e. by knowing all elements of
the universe $\mathbb{U}$ that are entities, along with their types, as well as all elements of $\mathbb{U}$ that are relations, along with their sources, targets and types.

**Example 5** For any entity $e$, the example query $\text{Isolated}(e)$ checks whether $e$ is contained in the graph model as an entity that has no incoming or outgoing relations. Formally, $G.\text{Isolated}(e).\text{out} = \{e\}$ if $G.\text{Query}(e :: *).\text{out} \neq \emptyset \land G.\text{Query}(e (\xrightarrow{\text{e}}) *).\text{out} = \emptyset = G.\text{Query}( (\xrightarrow{\text{e}}) e).\text{out}$, while $G.\text{Isolated}(e).\text{out} = \emptyset$ otherwise. Such a query might be useful to consult before deleting $e$ from the model.

### 2.1.5.2 Manipulation and delta

**Definition 19 (Graph manipulation operation)** The graph manipulation operation $\text{Mod}$ over metamodel $\text{MM} = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle$ is a graph access operation that is not a query, i.e. the resulting model may differ from the original graph model that operation is applied on.

Discussing how model manipulation operations can be specified is out of scope for the current thesis. As one example, graph transformation rules will be introduced in Section 2.2.4, but there is no assumption whatsoever that the model is allowed to change along graph transformation rules only.

The original and result graph models take their elements from the same universe: $\text{Elements}_{G}, \text{Elements}_{G'} \subset \mathbb{U}$. Consequently, there may be some entities or relations that are shared between them; others may be created by the manipulation operation (i.e. only appearing in the resulting model), yet others may be deleted (only present in the original model). In order to be able to handle the effect of such general manipulation operations, we need to introduce one more concept to express how $G'$ differs from $G$. The *delta* between two graph models is a set of graph element insertions and deletions that transforms the first model into the second. In other words, it is the difference between respective entity definitions and relation definitions (where such a definition includes the type, and in case of relations the source and target as well). For the sake of simplicity, element retyping, edge redirection, etc. are simply interpreted in the delta as the deletion of the old element and insertion of the updated version. In this case, we can use the notational shorthand $\lnot (e :: c)$ to represent that the instance $e$ of classifier $c$ was deleted, $+(e_s \xrightarrow{r::f} e_t)$ to represent that the instance $r$ of feature $f$ was inserted pointing from $e_s$ to $e_t$, and so on.

**Definition 20 (Graph delta)** The difference between two graph models, the pre-graph $G_1 = \langle \text{Ent}_1, \text{Rel}_1, \text{src}_1, \text{trg}_1, \text{typ}_1 \rangle$ and the post-graph $G_2 = \langle \text{Ent}_2, \text{Rel}_2, \text{src}_2, \text{trg}_2, \text{typ}_2 \rangle$, both conforming to the same metamodel $\text{MM}$, is defined to be $\delta = G_2 - G_1 := \langle \delta_\text{ent}, \delta_\text{rel}, \delta_\text{ent}^+, \delta_\text{rel}^+ \rangle$, containing entity deletions $\delta_\text{ent} = \{ \lnot (e :: c) | (e, c) \in G_1.\text{Query}( * :: * ).\text{out} \setminus G_2.\text{Query}( * :: * ).\text{out} \}$, relation deletions $\delta_\text{rel} = \{ \lnot (e_s \xrightarrow{r::f} e_t) | (r, f, e_s, e_t) \in G_1.\text{Query}( * :: * ).\text{out} \setminus G_2.\text{Query}( * :: * ).\text{out} \}$, entity insertions $\delta_\text{ent}^+ = \{ + (e :: c) | (e, c) \in G_2.\text{Query}( * :: * ).\text{out} \setminus G_1.\text{Query}( * :: * ).\text{out} \}$, and relation insertions $\delta_\text{rel}^+ = \{ + (e_s \xrightarrow{r::f} e_t) | (r, f, e_s, e_t) \in G_2.\text{Query}( * :: * ).\text{out} \setminus G_1.\text{Query}( * :: * ).\text{out} \}$. The set of all potential graph deltas over a metamodel $\text{MM}$ is $\text{Deltas}_{\text{MM}}$.

Note that $G_2$ can be reconstructed from $G_1$ and $\delta = G_2 - G_1$, while $G_1$ can be reconstructed from $G_2$ and $\delta$; these reconstruction operations are denoted as $G_1 + \delta$ and $G_2 - \delta$, resp. Finally, for model manipulation operation $\text{Mod}$, let $G.\text{Mod}.\delta$ denote the difference $G.\text{Mod}.r - G$. Due to these reconstruction properties, it would be possible to alternatively define manipulation operations.
as maps from source models to valid update deltas that can be added to the source model to obtain a result model, in the style of e.g. ASM [BS03].

A manipulation operation is called **elementary** if it has a constant, single-element delta: the insertion or deletion of a single entity or relation.

In attributed graphs, $Dat$ is required to be immutable, so manipulation operation $Mod$ is not permitted to have $G.Mod.\delta$ contain $\pm(e :: c)$ if $e \in Ent_{Dat}$, or $\pm(e \xrightarrow{(r:t)} e)$ if $r \in Rel_{Dat}$, these are additional preconditions to modification operations. The creation or deletion of value assignment edges ($r \in Rel_{val}$), however, is allowed and interpreted as updating the value of attributes.

**Example 6** For a Petri net transition $t$, $Fire(t)$ is a manipulation operation parameterized by a transition $t$ that transforms a Petri net model into the result of firing $t$ according to the transition firing semantics. The precondition of $G.Fire(t)$ is that $G$ contains $t$ as a transition and that this transition is fireable according to Petri net semantics. When applied on the sample instance model from Example 3 on page 13 (which will be referred to as $G_1$), the result $G'_1 = G_1.Fire(t_2).r$ is identical to $G_1$ except for the omission of marking $m_4$ and token $k_4$, as well as the addition of new token $k_5$ and new marking edge $m_5$ from $p_2$ to $k_5$. In other words, $G_1.Fire(t_2).\delta = \{- (m_4 :: Marking), \{- (p_3 \xrightarrow{(m_4::Marking)} m_5)\}\}$. Analogously, for the attributed metamodel and the instance model from Example 4 on page 17 denoted as $G_2$, the effect of the model manipulation is $G_2.Fire(t_2).\delta = \{\emptyset, \{- (p_2 \xrightarrow{(m_2::Marking)} 0), (m_3 :: Marking) 1\}, \emptyset, \{ (p_3 \xrightarrow{(m_4::Marking)} 0), (m_2 :: Marking) 1\}\}$.

### 2.1.6 Modeling paradigms

There are several industrial paradigms of modeling. They do not conform perfectly to the generic definitions of Section 2.1, but are similar to a degree. A few modeling frameworks are briefly described in the following, to illustrate how the theoretical concepts of this chapter translate into modeling practice.

Although the paradigms discussed below can be investigated and compared regarding numerous aspects, the focus here is on significant deviations from and extensions to the specific formalization of graph models and metamodels used in Section 2.1 and throughout the thesis. Some of their important properties are summarized by Table 2.2.

#### 2.1.6.1 MOF

One of the most well-known modeling ecosystems is the Meta Object Facility (MOF) [OMG03] by the Object Management Group (OMG). The flagship metamodel of OMG is the multi-purpose Unified Modeling Language (UML) [OMG11], whose instance models - and the associated intuitive concrete syntax diagrams - support various phases of object-oriented software development, such as requirements gathering, architecture, design, etc. Additional MOF-based standard metamodels include the Common Warehouse Metamodel (CWM) [PCTM02].

A model in MOF consists of **objects** (structural entities in terms of Section 2.1.2). They have attribute **slots** (value assignments) and are interconnected by potentially ordered **links** (structural relations). The association type determines whether links are **navigable** in one or both directions. In general, however, it is difficult to say which kind of elementary model queries are (efficiently) supported in addition to navigation, as there is no single standard implementation for representing MOF models.
CHAPTER 2. BACKGROUND

MOF metamodels can be thought of as an extension to the metamodel definition given in Section 2.1.2, with additional information such as the packaging of types or executable operations. The structural part of MOF metamodels define Classes (classes) and interconnecting Associations (associations), as well as Generalizations (supertyping). Some Classes can be designated as abstract, meaning that they can have no direct instances. Associations may be restricted in multiplicity. Some Associations are containments: they have one-to-many multiplicity and strictly enforce their instances to form an acyclic containment hierarchy. Class supertyping is allowed, but Association supertyping is not. The structural metamodel is complemented by the data algebra which is standardized but extensible by enumerations, etc. Finally as unidirectional glue there are Attributes (attribute names) characterized by a class, a datatype and multiplicity.

2.1.6.2 EMF

The Eclipse Modeling Framework (EMF) [EMF] by the Eclipse Foundation [ECLb] is similar to MOF, but it is more wide-spread in DSM applications, and therefore more important for this thesis.

An EMF model consists of EObjects (structural entities in terms of Section 2.1.2). EObjects have fields (value assignments) and are interconnected by ordered references (structural relations). Both kinds of relations are uniquely identified by feature, source EObject and target/value; “parallel edges” are not allowed, and the relation instance does not even exist as an individual Java object. Furthermore, relations can only be navigated in one direction (see Section 5.1.1 for further restrictions in basic model queries).

EMF uses Ecore metamodels to describe the abstract syntax of a modeling language. Ecore metamodels can be thought of as an extension to the metamodel definition given in Section 2.1.2, with additional information such as the packaging of types, or executable operations. The structural metamodel consist of EClasses (classes) and interconnecting EReferences (associations). EClasses may be abstract, forbidding direct instantiation. EReferences may be restricted in multiplicity. Two EReferences can be designated as each other’s EOpposite (inverse); this symmetric relationship will be
2.2. GRAPH PATTERNS AND GRAPH TRANSFORMATION

automatically maintained for their instance relations. Some EReferences are containments: they have one-to-many multiplicity and strictly enforce their instances to form an acyclic containment hierarchy. EClass supertyping is allowed, but feature supertyping is not. The structural metamodel is complemented by the data algebra which includes Java primitive values, enumerations, etc., for which one can define EDataTypes as "aliases" of these datatypes. Finally as unidirectional glue there are EAttributes (attribute names) characterized by an EClass, an EDataType and multiplicity.

More technical details on EMF are presented in Section 5.1.1.

2.1.6.3 VPM

The thesis also relies on the VPM (Visual and Precise Metamodeling) [VP03] metamodeling approach. VPM was chosen as the basis of the Viatra2 model transformation framework, as it is specifically designed to support transformation between heterogeneous modeling formalism, and thus offers a very rich and permissive modeling platform.

An VPM model consists of two kinds of model elements: Entity elements (structural entities in terms of Section 2.1.2) and Relations (structural relations). Entities are organized in a containment hierarchy (forest), and each of them also has a value, which is always a string. Relations are unordered and have a source and a target (and they are bidirectionally navigable); due to the flexibility of VPM, these Relation ends can be Relations as well as Entities. Elements of both kinds have a name that is string that must be unique within the namespace, which is the container (in case of Entities) or source (in case of Relations). VPM aims at efficient evaluation for each kind of elementary model query (see Section 2.1.5), regardless which parameters are wildcards.

The structural part of VPM metamodels is represented in VPM and does not necessarily conform to any specific meta-metamodel: it consists of Entities as classes and interconnecting Relations as associations. VPM supports multi-typing, i.e. each model element can have zero or more metamodel elements designated as its type (therefore typ actually maps to the power set of metamodel elements). Metamodel elements can be designated as final (analogous to abstract), so that they cannot be directly instantiated. Relations may be restricted in multiplicity. Some Relations are containments: their instances imply that the target is contained in the container (however, Entity instances may also contain each other without containment Relations), forming an acyclic containment hierarchy. Supertyping can be established both between Entity and between Relation elements. The structural metamodel is complemented by the data algebra which in this case is the set of character strings; all data values must be represented as a string. As unidirectional glue, there are two built-in attribute names (name and value, as already mentioned) and further ones can not be defined. This simplified description of the data algebra and attribute names omits some properties (e.g. multiplicity) only relevant in specifying metamodels.

2.2 Graph patterns and graph transformation

Having introduced core modeling concepts, there is a second important topic that is a prerequisite for the contributions of this work. Graph patterns, pattern matching and other notions will be defined in the following, that originate from the field of Graph Transformation (GT) [EEKR99].

2.2.1 Graph pattern basics

Graph patterns are one of the central notions used throughout the thesis. Although a constraint satisfaction problem-based definition is given below in the style of [VB07], it should be noted that
there are several alternative definitions in literature, such as “algebraic graph transformation” based graph morphisms and category theory [EEPT06].

**Definition 21 (Graph pattern)** A graph pattern \( P = \langle V, C \rangle \) over a metamodel \( MM = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle \) contains a set of pattern variables \( V \), and a set of graph constraints \( C = C^\text{ent} \cup C^\text{rel} \cup C^= \cup C^\neq \) attached to them. \( V \) is partitioned into entity variables \( V^\text{ent} \) and relation variables \( V^\text{rel} \). Constraints can be of the following kinds:

- **Entity constraints** \( C^\text{ent} \subseteq V^\text{ent} \times \text{Cls} \) state that a variable is a node of a certain type.
- **Relation constraints** \( C^\text{rel} \subseteq V^\text{ent} \times V^\text{rel} \times V^\text{ent} \times \text{Fea} \) state that a variable is an edge of a certain type, connecting two given variables representing the source and the target of the edge.
- **Equality constraints** \( C^= \subseteq V \times V \) state that two variables represent the same element.
- **Inequality constraints** \( C^\neq \subseteq V \times V \) state that the two variables represent different elements.

To identify the variables and constraints of a specific pattern \( P \), we use \( V(P) \) and \( C(P) \), respectively. The set of all graph patterns over a metamodel is denoted as \( \text{Patterns}_{MM} \).

More advanced formalisms such as the pattern language [VB07] of the Viatra2 tool may define additional kinds of pattern constraints, some of which will be discussed later in this thesis.

**Definition 22 (Substitution of graph pattern)** A partial substitution \( s : V \rightarrow \text{Elements}_G \) of a graph pattern \( P = \langle V, C \rangle \) in a graph model \( G = \langle \text{Ent}, \text{Rel}, \text{src}, \text{trg}, \text{typ} \rangle \) conforming to a metamodel \( MM = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle \) is a partial function that maps some variables of the pattern to graph elements. In particular, entity variables \( V_{\text{Ent}} \) are mapped to \( \text{Ent} \) and relation variables \( V_{\text{Rel}} \) are mapped to \( \text{Rel} \). Let \( s(v) \in \text{Elements}_G \) denote the model element assigned by \( s \) to the variable \( v \in V \). Let \( \text{Dom}(s) = \{ v \mid \exists e \in \text{Elements}_G : s(v) = e \} \) be the set of variables that have substituted values. A partial substitution \( s : V \rightarrow \text{Elements}_G \) is a substitution if \( \text{Dom}(s) = V \), i.e. it provides assignments for all variables of the pattern. If an order of variables \( \langle v_1, v_2, \ldots, v_k \rangle \) is fixed, any substitution \( s \) can be uniquely represented by the tuple \( \langle s(v_1), s(v_2), \ldots, s(v_k) \rangle \).

**Definition 23 (Domain of constraints)** For pattern \( P = \langle V, C \rangle \), the domain \( \text{Dom}(c) \) of constraint \( c \in C \) is the set of variables involved in the constraint:

- For entity constraint \( c = \langle v, t \rangle \in C^\text{ent} \), \( \text{Dom}(c) = \{ v \} \).
- For relation constraint \( c = \langle a, v, b, t \rangle \in C^\text{rel} \), \( \text{Dom}(c) = \{ a, v, b \} \).
- For equality constraint \( c = \langle u, v \rangle \in C^= \), \( \text{Dom}(c) = \{ u, v \} \).
- For inequality constraint \( c = \langle u, v \rangle \in C^\neq \), \( \text{Dom}(c) = \{ u, v \} \).

**Definition 24 (Constraint satisfaction)** For a given substitution \( s \) of a graph pattern \( P = \langle V, C \rangle \), any constraint \( c \in C \) is either satisfied by \( s \) or not:

- The substitution \( s \) satisfies an entity constraint \( c = \langle v, t \rangle \in C^\text{ent} \) iff \( s(v) : t \).
- The substitution \( s \) satisfies a relation constraint \( c = \langle a, v, b, t \rangle \in C^\text{rel} \) iff \( \text{src}(s(v)) = s(a) \) and \( \text{trg}(s(v)) = s(b) \) and \( s(v) : t \).
• The substitution $s$ satisfies an equality constraint $c = \langle u, v \rangle \in C = \iff s(u) = s(v)$.

• The substitution $s$ satisfies an inequality constraint $c = \langle u, v \rangle \in C \neq \iff s(u) \neq s(v)$.

Satisfaction can be defined in the same way if $s$ is a partial substitution with $\text{Dom}(s) \supseteq \text{Dom}(c)$.

**Definition 25 (Graph pattern match)** A match $m : V \rightarrow \text{Elements}_{G}$ is a substitution of the variables $V$ of $P$ that satisfies all constraints $c \in C$ of $P$. This will be denoted as $G, m \models P$. The match set $\text{MatchSet}_{G}^{P} = \{ m \mid G, m \models P \}$ is the set of all matches of a pattern in a graph model.

Remark: from now on, a single metamodel $MM = \langle \text{Cls}, \text{Fea}, \text{owner}, \text{range}, \text{super} \rangle$ is assumed without loss of generality, and the metamodel will often be omitted from further definitions.

**Example 7** Our first graph pattern marked is defined over the purely structural metamodel in Example 2 on page 11 and it identifies places marked by tokens. The pattern is visualized in Figure 2.7.

![Figure 2.7: A sample graph pattern capturing marked places](image)

The pattern variables are $P$, $M$ and $K$. $P$ and $K$ are entity variables (depicted as boxes within the pattern), while $M$ is a relation variable. Contrast $c_{1} = \langle P, \text{Place} \rangle \in C_{\text{ent}}$ claims that the image of $P$ should instantiate Place, and is depicted as the type name within the box of $P$, delimited by a colon. Similarly, $c_{2} = \langle K, \text{Token} \rangle \in C_{\text{ent}}$ claims that the image of $K$ should instantiate Token. Finally, the relation constraint $c_{3} = \langle P, M, K, \text{Marking} \rangle \in C_{\text{rel}}$ states that the image of $M$ should be a relation that instantiates Marking, has the image of $P$ as its source and the image of $K$ as its target; it is visualised as an arrow from $P$ to $K$, labeled by variable name $M$ and type name Marking delimited by a colon.

Considering the graph model of Figure 2.3 and Example 3 on page 13, the pattern will have four matches altogether, as displayed by Figure 2.8. There is one match for each token: $m^{(1)} = \{ P \mapsto p_{1}, M \mapsto m_{1}, K \mapsto k_{1} \}$, $m^{(2)} = \{ P \mapsto p_{1}, M \mapsto m_{2}, K \mapsto k_{2} \}$, $m^{(3)} = \{ P \mapsto p_{1}, M \mapsto m_{3}, K \mapsto k_{3} \}$ and $m^{(4)} = \{ P \mapsto p_{1}, M \mapsto m_{4}, K \mapsto k_{4} \}$. If the order of variables is defined as $\langle P, M, K \rangle$ (as indicated in the header of Figure 2.7), matches can be denoted as $\langle p_{1}, m_{1}, k_{1} \rangle$ and so on.

### 2.2.2 Complex graph patterns

#### 2.2.2.1 Negative application conditions

A negative application condition (NAC) prescribes contextual conditions that, if satisfiable, invalidate a match of the pattern.

**Definition 26 (Graph pattern with negative application condition)** A pattern with NAC is $PN = \langle P, N^{*} \rangle$ where $P = \langle V, C \rangle$ is a (positive) graph pattern, and $N^{*}$ is a set of negative application conditions $N = \langle V_{i}, C_{i} \rangle$, each being a graph pattern, such that $P \subseteq N_{i}$ (meaning that $V \subseteq V_{i}$ and $C \subseteq C_{i}$).
Commonly, only the subpattern $\tilde{N}_i = N_i \setminus P$ is explicitly indicated and depicted in figures and code extracts, which is defined as $\tilde{N}_i = (\tilde{V}_i, \tilde{C}_i)$, where $\tilde{C}_i = C_i \setminus C$ and $\tilde{V}_i \subseteq V_i$ is the set of variables involved in $\tilde{C}_i$.

**Definition 27 (Match of graph pattern with NAC)** A match $m : V \rightarrow \text{Elements}_G$ of $PN = (P, N^*)$ in graph model $G$ is a match of the positive pattern $G, m \models P$, where there is no $N_i \in N^*$ and match $m_i : N_i \rightarrow G$ such that $m \subseteq m_i$ (meaning that $m_i(v) = m(v)$ for all $v \in \text{Dom}(m) = V$ variables of $P$).

Some pattern formalisms [Ren04a, VB07] even permit NACs to have NACs of their own. In fact, it is possible to formalize NACs as special pattern constraints $\tilde{N}_i \in C^{NAC}$ (see also Section 2.2.2.3), eliminating the need for a distinct concept of graph pattern with NACs. If there is no limit on the number of negations that can be nested within each other, graph patterns (without attribute constraints) become expressively equivalent to first order formulae over the predicates describing the graph model [Ren04b].

**Example 8** An unmarked place, i.e. a place without tokens, can be expressed with pattern unmarked, which is a graph pattern with NAC (see Figure 2.9). The positive pattern has a single variable $P$ and a single entity constraint $c_1 = \langle P, \text{Place} \rangle \in C^{ent}$. There is a single negative application condition $N_1$, which is the pattern marked defined before in Example 7. Overall, the pattern will match Place instances for which there are no corresponding Token instances with which they would form a match of marked; in other word, tokenless places. In the graph model of Figure 2.1(a) and Example 3, the pattern will have a single match $\langle p_2 \rangle$.

Figure 2.10 shows the alternative, more concise notation where the negative application condition is represented by a negative subpattern $\tilde{N}_1$ instead of the full NAC pattern $N_1$. $\tilde{N}_1$ is essentially the difference between the negative pattern and the positive one. Here the duplication of $P$ and its entity constraint is avoided.

### 2.2.2.2 Attributed graph patterns

Next the structure of graph patterns over attributed models is discussed.

**Definition 28 (Graph pattern with attributes)** In case of attributed metamodels, $V^{ent}$ of a graph pattern is further partitioned into structural and data entity variables ($V^{ent}_{Str}$ and $V^{ent}_{Dat}$); similarly $V^{rel}$ is partitioned into three sets: structural relations $V^{rel}_{Str}$ between structural elements, value assignments
2.2. GRAPH PATTERNS AND GRAPH TRANSFORMATION

Positive variables

Positive constraints

NAC variables

NAC constraints

Figure 2.9: Graph pattern with NAC capturing unmarked places

Figure 2.10: Graph pattern with NAC capturing unmarked places (concise version)

V_{Val} \bar{V}_{Val} that connect structural elements to their attribute values, and data relations V_{Dat} between attribute values. Each of these partitions have a corresponding constraint type:

- Class constraints C^{ent}_{Str} \subseteq V^{ent}_{Str} \times Cls_{Str} express that a variable represents a structural entity of a certain type;
- Datatype constraints C^{ent}_{Dat} \subseteq V^{ent}_{Dat} \times Cls_{Dat} assert attribute types such as integer;
- Association constraints C^{rel}_{Str} \subseteq V^{rel}_{Str} \times V^{rel}_{Str} \times V^{rel}_{Str} \times Fea_{Str} express that a variable represents a structural edge of a certain association type;
- Attribute assignment constraints C^{rel}_{Val} \subseteq V^{ent}_{Str} \times V^{rel}_{Val} \times V^{ent}_{Dat} \times Fea_{Val} mean that a certain value assignment, associated with an attribute name, links a structural variable to a data variable as its attribute value;
- Data predicate constraints C^{rel}_{Dat} \subseteq V^{ent}_{Dat} \times V^{rel}_{Dat} \times V^{ent}_{Dat} \times Fea_{Dat} express that the variable represents a data relation instantiating a data predicate between given data entities; this essentially means an attribute constraint check among the variables corresponding to attribute values.

The reader is referred to Table 2.1 on page 9 for an overview of these variables and constraints and their relationship to model element kinds.

It is possible for graph patterns to reference the data algebra Dat, since Dat is known a priori unlike the structural instance models. This gives rise to a new kind of pattern constraint:

Definition 29 (Value literal pattern constraint) In an attributed graph pattern P = ⟨V, C⟩ over data algebra Dat, literal constraints C^{lit} \subseteq V^{ent} \times Ent_{Dat} state that the given variable must represent
an a priori known constant data value. A substitution \( s \) satisfies a literal constraint \( c = (v, x) \in C^{lit} \) iff \( s(v) = x \). \( \text{Dom}(c) = \{v\} \).

Value literal constraints allow a pattern to capture data entries in variables; further structural or data variables may be connected to these variables via constraints.

**Example 9** Over the alternative attributed metamodel of Example 4 on page 17, the attributed version of graph pattern unmarked (see Figure 2.11) can be defined as follows: the three variables are structural entity variable \( P \in V_{\text{Str}} \) (depicted as a solid box), value assignment variable \( M \in V_{\text{Val}} \) (depicted as a dashed arrow), and data entity variable \( Z \in V_{\text{Dat}} \) (depicted as a dashed box). The type constraints are \( c_1 = (P, \text{Place}) \in C_{\text{Str}} \), \( c_2 = (Z, \text{Integer}) \in C_{\text{Val}} \), and \( c_3 = (P, M, Z, \text{Marking}) \in C_{\text{Rel}} \). So far, these constraint ascertain that \( P \) is a place having \( Z \) tokens. Finally the literal constraint \( c_4 = (Z, 0) \in C^{lit} \) (depicted as a white oval within the box of \( Z \)) identifies \( Z \) as the data value 0. Overall, the pattern matches a place \( p \) iff \( G | p.\text{Marking} = 0 \), i.e. it has no tokens. Now it is evident how this pattern is functionally equivalent to the version of unmarked in Example 8 on page 26, that was specified in context of a pure structural metamodel.

**Example 10** The related marked pattern is shown in Figure 2.12. There are now two variables with Integer datatype constraints: \( Z \) and \( NZ \). \( NZ \) is the marking of \( P \), while \( Z \) is identified with 0 as before. An inequality constraint \( c_6 = (Z, NZ) \in C^{\neq} \) (depicted as a floating oval connected to both variables) states that \( Z \) and \( NZ \) should be mapped to different elements. (A possible alternative would include a data relation variable and a data relation constraint of type \( > \) from \( NZ \) to \( Z \) instead of the inequality constraint; the effect would be the same since markings in reachable states are always non-negative.) Overall, the pattern identifies places with non-zero (respectively positive) marking; and is therefore functionally equivalent to the version of marked in Example 7.
There is a final issue that requires attention. Including \( \text{Dat} \) in the graph model raises new kinds of problems for pattern matching, as the entirety of the infinite data algebra \( \text{Dat} \) cannot be manifested (i.e. enumerated, stored in memory). More precisely, some patterns cannot be matched by a finite matcher; a trivial example would be the pattern \( P = \langle V, C \rangle \) with single variable \( v \in \text{Dat} \) and constraint \( c \in C_{\text{Dat}}^\infty \) where \( c = \langle v, \text{Number} \rangle \); a pattern matcher would have to enumerate all numbers to compute the match set of the pattern, which is impossible to perform with finite resources. A more complex example would be the marked pattern in Example 10 if we omitted the value literal constraint on \( Z \): the matcher would have to enumerate all integer values for variable \( Z \) that are different from \( \text{NZ} \). Naturally there are other kinds of mathematical reasoning that could be performed to obtain a finite characterization of the match set of such a pattern; however, graph pattern matchers are not aimed to be equation solvers. To avoid such problems, we have to identify those attributed patterns that can be matched by a pattern matcher without requiring an equation solver or other mathematical reasoning; for this purpose we distinguish assignable variables.

**Definition 30 (Assignable entity variable)** Structural entity variables are always assignable. A data entity variable is assignable iff

- it has an associated value literal pattern constraint, therefore its value is known; or

- appears as the target variable of a value assignment constraint, thus it is available as an attribute value of a model element from the structural model; or

- it is determined by a data predicate that is enumerable (see Definition 18) by assignable variables, thus it can be computed from other variables that are already substituted. Here functional or otherwise computable relationships (e.g. division) are asserted by appropriate data predicate constraints (using either the hypergraph or the auxiliary node representation).

**Definition 31 (Matchable attributed graph pattern)** A graph pattern is matchable iff each entity variable \( v \in V_{\text{ent}} \) is assignable, and any NACs (as well as called patterns, to be introduced in Section 2.2.2.3) are also matchable patterns.

The current thesis focuses on graph pattern matching, as opposed to integer-domain constraint solving, (in)equation solving, etc., therefore all graph patterns will be assumed to be matchable.

### 2.2.2.3 Pattern composition

The next advanced feature will help us build more complex patterns. Composing (calling) previously defined graph patterns as a kind of hyperedge pattern constraint [HVVO7] can improve the conciseness of the language.

**Definition 32 (Pattern composition constraint)** In a composite graph pattern \( P = \langle V, C \rangle \), a pattern composition constraint for a called pattern \( P_{\text{called}} \) is \( c_{\text{call}} = \langle P_{\text{called}}, \text{composer} \rangle \), where \( \text{composer} : V \rightarrow V_{\text{called}} \) is a (partial) mapping from the variables \( V \) of the composite pattern to variables \( V_{\text{called}} = V(P_{\text{called}}) \) of the called pattern. The composition constraint states that matches of the composite pattern must be aligned with a match of the called pattern. A substitution \( s \) satisfies a pattern composition constraint \( c = \langle P_{\text{called}}, \text{composer} \rangle \in C_{\text{call}} \) iff \( \text{incidents} \neq \emptyset \), where \( \text{incidents} = \{ m \mid G, m \models P_{\text{called}} \land \forall v \in \text{Dom}(\text{composer}) s(v) = m(\text{composer}(v)) \} \). \( \text{Dom}(c) = \text{Dom}(\text{composer}) \).
CHAPTER 2. BACKGROUND

Figure 2.13: Composite graph pattern capturing transitions disabled by unmarked incoming places

Note that NACs can alternatively be defined as a pattern constraint that is analogous to pattern composition, by simply negating the truth value of the satisfaction condition in the previous definition: \( c_i \in C^{NAC} \) is satisfied by \( s \) iff \( \nexists m : G, m \models N_i \land \ldots \). Negative composition is sometimes used as an alternative name for NACs.

**Example 11** The conciseness of pattern composition is demonstrated by pattern `unmarkedDisables` (see Figure 2.13) that identifies transitions that cannot be fired because an unmarked place is connected by an InArc. The `unmarked` pattern can clearly be reused here. Pattern `unmarkedDisables` uses variables \( P, I, \) and \( T \), with respectively a relation constraint and an entity constraint for the latter two. The composition constraint \( c_3 = \langle \text{unmarked}, \text{composer}_1 \rangle \in C^{call} \) calls the `unmarked` pattern, with \( \text{composer}_1 \) (not detailed in the figure) mapping the variable \( P \) to the variable in pattern `unmarked` with the same name; in essence it states that \( P \) is an unmarked Place. In the graph model of Figure 2.1(a) on page 10, the composite pattern will have no matches; however, \( \langle p_3, i_3, t_2 \rangle \) and \( \langle p_3, i_2, t_1 \rangle \) would become valid matches if \( t_2 \) was fired (Figure 2.1(c)).

**Discussion on cyclic and acyclic composition.** Let us now assume that pattern compositions are acyclic. In this case it can be shown that pattern composition is mainly a syntactic sugar. For acyclic composition, we can always (recursively) embed the variables and constraints of the called pattern into the composite pattern by applying \( \text{composer}^{-1} \) as “variable renaming”. This process, called pattern flattening [HV07], transforms a composite graph pattern and its called patterns into a single pattern with equivalent semantics but without composition. It immediately follows that the expressiveness of the pattern language is not affected by the composition language element, as long as recursive pattern calls are disallowed.

With strictly acyclic composition, expressiveness is equivalent to first-order logic [Ren04b]. With recursion, however, some higher-order properties such as transitive closure become expressible.

Pattern composition will be assumed throughout the thesis to be acyclic, because there are no known methods that can efficiently and incrementally evaluate such recursive queries. There are non-incremental techniques such as magic sets [BMSU86], with application for graph pattern matching [VHV08, HJG08], capable of dealing with recursive pattern composition; supported, for instance, by Viatra2.

Unfortunately, recursion significantly increases the difficulty of incremental maintenance (see Section 3.2.3). Simpler incremental techniques designed for the non-recursive case are prone to fail, due to the fact that a fixpoint operator would be required to unambiguously define the match set of recursive queries in first-order logic (see also Section 4.2.3, page 84). Recursive patterns therefore remain future work, and will not be addressed in general by this thesis. However, in many practi-
2.2. GRAPH PATTERNS AND GRAPH TRANSFORMATION

In practical cases, the typical purpose of recursion is to express some sort of transitive closure, for which a specialized solution is provided in Section 4.2.

2.2.2.4 Aggregate constraints

Aggregation is a useful capability that has its roots in pattern composition and lets the pattern matcher aggregate the matches of a called pattern into a single value that will be used in a match of the aggregating pattern. Example use cases include counting matches, summing up variables, taking the maximum, etc.

Definition 33 (Match aggregate constraint) In a aggregating graph pattern \( P = \langle V, C \rangle \), a match aggregate constraint for a called pattern \( P_{\text{called}} \) is \( \text{c}^{\text{aggregate}} = \langle P_{\text{called}}, \text{composer}, \text{aggregator}, v_{\text{result}} \rangle \), where

- \( \text{composer} : V \to V_{\text{called}} \) is a (partial) mapping from the variables \( V \) of the aggregating pattern to variables \( V_{\text{called}} = V(P_{\text{called}}) \) of the called pattern,
- \( \text{aggregator} = \langle \text{mapper}, \text{reducer}, \text{unmapper} \rangle \) describes how matches should be aggregated using the following elements,
  - \( \text{reducer} = \langle \mathbb{A}, \otimes \rangle \) is an Abelian group that will reduce (aggregate) the result,
  - \( \text{mapper} \) is a function that maps matches of \( P_{\text{called}} \) to the reducer group \( \mathbb{A} \),
  - \( \text{unmapper} \) is a function that maps an element of the reducer group \( \mathbb{A} \) to \( \text{Ent}_\text{Dat} \cup \{ \bot \} \), where \( \bot \not\in U \) is a special value indicating that aggregation is not possible,
- \( v_{\text{result}} \) is a data entity variable for storing the result, with \( v_{\text{result}} \not\in \text{Dom}(\text{composer}) \).

The aggregate constraint states that in matches of the aggregating pattern, the result variable must be the aggregate of all incident matches of the called pattern, by mapping said matches first by \( \text{mapper} \), then taking their product in \( \text{reducer} \) to reduce them to a single value, and then applying \( \text{unmapper} \). A substitution \( s \) satisfies a match aggregate constraint \( \text{c}^{\text{aggregate}} = \langle P_{\text{called}}, \text{composer}, \text{aggregator}, v_{\text{result}} \rangle \in C^{\text{aggregate}} \) with \( \text{aggregator} = \langle \text{mapper}, \text{reducer}, \text{unmapper} \rangle \) iff \( m(v_{\text{result}}) = \text{unmapper}(\prod^\otimes_{m \in \text{incidents}} \text{mapper}(m)) \), where \( \text{incidents} = \{ m \mid G, m \models P_{\text{called}} \land \forall v \in \text{Dom}(\text{composer}) s(v) = m(\text{composer}(v)) \} \). The \( \otimes \)-product used in this condition is well-defined since \( \otimes \) is associative and commutative. \( \text{Dom}(c) = \text{Dom}(\text{composer}) \cup \{ v_{\text{result}} \} \).

Note that it follows from the definition that if \( \text{unmapper} \) maps the \( \otimes \)-product to \( \bot \), then the constraint cannot be satisfied by any substitution \( s \), since \( s(v_{\text{result}}) \not\in U \) is required, so the aggregating pattern will not match.

In analogy with data relation constraints, an aggregate constraint is not always enumerable, but functionally determined by variables \( \text{Dom}(\text{composer}) \). In the special case of \( \bot \) discussed above, the set of enumerated variable substitutions would be empty.

Example 12 A common use case for the aggregate constraint is counting the matches of the called pattern that are incident on variables of the aggregating pattern. This is achieved by \( \text{aggregator}_{\text{count}} = \langle \text{mapper}_{\text{count}}, \text{reducer}_{\text{count}}, \text{unmapper}_{\text{count}} \rangle \) where \( \text{reducer}_{\text{count}} = \langle \mathbb{Z}, + \rangle \) is the group of integers w.r.t. addition, \( \text{mapper}_{\text{count}} = m \mapsto 1 \) maps all matches to the integer value 1,
and \text{unmapper}_{\text{count}} \) is the identity (where \( \mathbb{Z} \subseteq \text{Ent}_{\text{Dat}} \) is assumed). This counting aggregator works by taking the number 1 for each incident match of the called pattern; then taking their “product” along the group operation, which is addition in this case, yielding the number of incident matches that were considered; and finally storing this result value in the result variable.

The pattern constraint-based formalization of NACs, mentioned earlier, can be thought of as a special case of counting aggregate, where the result variable (i.e. the number of incident matches of the called pattern) is constrained to be the value literal 0.

**Example 13** A further common example is summing up a numerical data value over matches of a called pattern that exposes this number as a parameter variable \( v \). \( \text{aggregator}_{\text{sum}(v)} = (\text{mapper}_{\text{sum}(v)}, \text{reducer}_{\text{sum}(v)}, \text{unmapper}_{\text{sum}(v)}) \) where \( \text{reducer}_{\text{sum}(v)} = (\mathbb{R}, +) \) is the group of (real) numbers w.r.t. addition and integers w.r.t. addition, \( \text{mapper}_{\text{sum}(v)} = m \mapsto m(v) \) maps all incident matches to the value they assign to their parameter \( v \), and \( \text{unmapper}_{\text{sum}(v)} \) is once again the identity (where \( \mathbb{R} \subseteq \text{Ent}_{\text{Dat}} \) is assumed).

**Example 14** To show an example where \( \text{unmapper} \) is not the identity function, consider averaging the value of a parameter \( v \) of the called pattern over incident matches of the called pattern. \( \text{aggregator}_{\text{avg}(v)} = (\text{mapper}_{\text{avg}(v)}, \text{reducer}_{\text{avg}(v)}, \text{unmapper}_{\text{avg}(v)}) \) where \( \text{reducer}_{\text{avg}(v)} = (\mathbb{R}, +) \times (\mathbb{Z}, +) \) is the Cartesian product group of (real) numbers w.r.t. addition and integers w.r.t. addition, \( \text{mapper}_{\text{avg}(v)} = m \mapsto (m(v), 1) \), and finally the actual averaging: \( \text{unmapper}_{\text{avg}(v)} = (\text{sum, count}) \mapsto \left\{ \begin{array}{ll} \text{sum}/\text{count} & \text{if count} \neq 0 \\ \bot & \text{if count} = 0 \end{array} \right\} \) (where \( \mathbb{R} \subseteq \text{Ent}_{\text{Dat}} \) is assumed).

### 2.2.2.5 Disjunction

Finally, disjunctive graph patterns allow multiple ways to be satisfied.

**Definition 34 (Graph pattern with disjunction)** A disjunctive pattern is \( PD = (V, PN^*) \) where \( V \) is the set of variables of the disjunctive pattern, and \( PN^* \) is a set of graph patterns (called pattern bodies) \( PN_j = (P_j, N_j^*) \), having (positive) variables \( V_j = V(PN_j) \) such that \( V \subseteq V_j \).

To distinguish from the variables of the pattern bodies, \( V \) is also called parameter variables or pattern header variables.

**Definition 35 (Match of graph pattern with disjunction)** A match \( m : V \rightarrow \text{Elements}_1(G) \) of \( PD = (V, PN^*) \) in graph model \( G \) is a substitution \( m : V \rightarrow \text{Elements}_1(G) \) that can be extended to a match of at least one of the pattern bodies: \( \exists PN_j \in PN^*, m_j \supseteq m : N_j \models PN_j \).

From now on, disjunctive, attributed patterns with NACs are allowed whenever graph patterns are used, unless otherwise noted.

**Example 15** The disjunctive pattern \text{disables} (see Figure 2.14) identifies a transition disabled by a place. There can be two reasons for this: either the place is unmarked and connected by an \text{InArc}, or is marked but connected by an inhibitor arc. These two cases (the first of which is already introduced as \text{unmarkedDisables}) form the two pattern bodies. The disjunctive pattern exposes variables \( P \) and \( T \), found in both bodies. Body-specific variables like \( H \) do not have a corresponding variable in the other body, therefore they cannot be parameters. In the graph model of Figure 2.1(a), the disjunctive pattern will have no matches; however, \( [p_2, g_1, t_1], [p_3, t_3, t_2] \) and \( [p_3, t_2, t_1] \) would become valid matches if \( t_2 \) was fired.
2.2. GRAPH PATTERNS AND GRAPH TRANSFORMATION

Parameter variables

\[ P \quad T \quad V \quad \text{Ent} \quad V \quad \text{rel} \quad V \quad \text{Ent} \quad V \quad \text{rel} \quad V \quad \text{Ent} \]

Body

Body variables

\[ \text{Body}_1 \quad \text{Body}_2 \]

Body constraints

\[ c_1 \quad C^{\text{rel}} \quad (T, \text{Transition}) \]
\[ c_2 \quad C^{\text{rel}} \quad (P, I, T, \text{InArc}) \]
\[ c_3 \quad C^{\text{call}} \quad (\text{unmarked}, \text{composer}_1) \]
\[ c_4 \quad C^{\text{rel}} \quad (T, \text{Transition}) \]
\[ c_5 \quad C^{\text{rel}} \quad (P, H, T, \text{InhibitorArc}) \]
\[ c_6 \quad C^{\text{call}} \quad (\text{marked}, \text{composer}_2) \]

Figure 2.14: Disjunctive graph pattern capturing disabled transitions

2.2.3 Graph pattern matching

For a software exerting rule-based behavior specified by graph transformation rules, as well as any other application based on graph patterns, a graph pattern matcher (PM) is a component of key importance. Given a graph model conforming to a metamodel and a set of graph patterns over the same metamodel, the role of the pattern matcher is to compute the set of matches of each graph pattern. In other words, it exposes the following query operation:

**Definition 36 (Graph pattern matching)** For graph pattern \( P \in \text{Patterns}_{MM} \) over a metamodel \( MM \), graph pattern matching \( \text{Match}(P) \) is a query operation that, when executed on model \( G \in \text{Graphs}_{MM} \), yields the output \( G.\text{Match}(P).\text{out} = \text{MatchSet}_{P}^{G} \) consisting of all matches of \( P \) in \( G \).

As query results were specified to be sets of tuples, this definition assumes the tuple-based representation of matches at an arbitrary but fixed ordering of variables.

In some cases, only a specific subset of matches are of interest, that are incident on a given set of elements. Therefore (similarly to the elementary query operations defined in Section 2.1.5) an advanced pattern matcher may offer query operations \( \text{Match}(P, \text{Input}) \) where some of the parameter variables \( V \) of the graph pattern \( P \) may be specified as a concrete graph element in model \( G \) using a partial substitution \( \text{Input} : V \to G \). This generalization is called seeded pattern matching, and the resulting filtered match set \( \text{ResultSet}_{G}^{P}(\text{Input}) = \{ m | m \in \text{MatchSet}_{P}^{G} \land \forall v \in \text{Dom}(\text{Input}) : m(v) = \text{Input}(v) \} \) is its result set (whereas the match set \( \text{MatchSet}_{G}^{P} \supseteq \text{ResultSet}_{G}^{P}(\text{Input}) \) will always refer to all matches of a pattern).

**Performance** is a serious issue in any system that offers graph pattern matching. It is easy to see that even for finite graph models, a graph pattern with only entity constraints can have a match set size up to \(|\text{Ent}|^{V}|\) that is polynomial in model size and exponential in pattern size; this is a lower bound for the time cost of pattern matching. Even if additional constraints (such as \( C^{\text{rel}} \)) reduce the match set size, graph pattern matching is still NP-complete [Roz97] in the size of the graph pattern. However, in our use cases, graph patterns can be considered fixed, and only the graph model is considered as “input”; matching a given pattern on models is only polynomial in model size. While in software engineering practice graph patterns (and thus the order of these polynomials) rarely grow very large, and the constraints are rarely permissive enough to manifest the worst-case
cost, it is a general practical observation that pattern matching is still the most resource-consuming phase of model transformation.

The most common way of performing graph pattern matching are local search (LS) pattern matching techniques that traverse a search tree of partial substitutions according to a search plan [VHV08]. An alternative approach will be introduced in Chapter 3.

**Example 16** Take for example the pattern fireable (see Figure 2.15) that identifies the fireable transitions of the Petri-net (i.e. those that are not disabled by any input place), relying on the pattern disables defined earlier in Example 15. If $G_1$ is the graph model of Figure 2.1(a) and Example 3, the application of the pattern matching operation would yield $G_1 Match(fireable).out = \{⟨t_1⟩, ⟨t_2⟩\}$, as both transitions are enabled. If $Fire(t)$ is the manipulation operation of Example 6 on page 21 that fires the given transition (precondition: $t$ is in the match set of fireable), and $G_1.Fire(t_2)$ leads to the new state $G'_1$, then $G'_1 Match(fireable).out = \emptyset$ as both transitions will become disabled.

### 2.2.4 Graph transformation rules

The mathematical formalism of Graph Transformation (GT) [EEKR99] provides a high-level rule and pattern-based manipulation language for graph models. This formalism is often preferred for its powerful behavior analysis techniques such as critical pairs [Plu93, LEO06]; and can be used for various purposes, including to specify model transformations (see e.g. [Var04], whose formalization will be loosely followed here).

**Definition 37 (Graph transformation rule)** A graph transformation rule $GTR = \langle LHS, RHS \rangle$ is specified by two graph patterns: a precondition (or left-hand side) pattern (with NAC) $LHS$ defining the applicability of the rule, and a postcondition (or right-hand side) positive pattern $RHS$ which declaratively specifies the result model after rule application. The variable sets of $LHS$ and $RHS$ are allowed to intersect\(^3\).

When the rule is applied on a match of the LHS, elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged. Without bothering with precise and fundamental formalization (which is available e.g. in [EEPT06]), the following definition expresses the essence of GT rule application.

**Definition 38 (Application of graph transformation rule)** The application of graph transformation rule $GTR = \langle LHS, RHS \rangle$ on a match $m : Elements(G) \to Elements_G$ in a graph

---

\(^3\)or, alternatively, the variable sets are disjoint, but connected through a mapping
model $G$ is the model manipulation operation $ApplyRule(GTR,m)$, defined here by its effect $G.ApplyRule(GTR,m).\delta$:

- **Deletion.** For each $v \in V_{LHS} \setminus V_{RHS}$, the model element $m(v)$ is deleted.

- **Insertion.** For each $v \in V_{RHS} \setminus V_{LHS}$, a new model element is created and assigned to $v$ in a way that $RHS$ becomes satisfied (i.e. the new element is created in a type-conforming and structurally consistent way).

If $LHS$ has no matches in $G$, then $GTR$ is not applicable.

**Semantic variations.** There are some semantic variation points in different GT formalisms. For example, in valid graph models, the source and target of a relation must be an entity of the graph; therefore a GT rule that deletes an entity must either also delete incident ("dangling edge") relations as a side effect, or impose a precondition that no such dangling relations exist.

A further issue is a potential conflict between deletion operations and insertion/preserving, which may arise if some pattern variables are mapped to the same model elements in a match. One of the solutions is to assign some kind of semantics to such rule execution, e.g. deletion overrides preservation. An alternative one is to impose the precondition that the match must be "injective" (different variables must be mapped to different elements). As a compromise between the two approaches, it is possible to allow mapping variables to the same value as long as certain identification conditions are met, which essentially state that the postcondition must impose compatible actions on them.

There are also variations which allow more refined change operations, such as redirecting existing relations, or changing the types of existing elements.

For the sake of determinism, $RHS$ is typically required to be non-disjunctive. Some pattern constraints that cannot be directly enforced by the model manipulation operations creating the image of the $RHS$, such as inequality or NACs, may either be disallowed in the $RHS$, or interpreted as a postcondition check.

Note that this core formalism of GT rules only defines which rules are applicable (and what happens if they are applied); it does not define e.g. how to react to changes of the model. It is up to implementations and extended formalisms to specify whether any GT rules should be applied at a given time, and which rule to apply on which match if there are several options.

**Example 17** The graph transformation rule $removeToken$ (see Figure 2.16) over the purely structural metamodel of Example 2 is applicable on a marked place (hence reusing as LHS the marked pattern defined in Example 7), the effect of application is the removal of one token from that place. The rule behaves so because variables $M$ and $K$ are only present in the LHS, therefore the token and the marking relation will be deleted. No new elements will be created, since $V_{place} \setminus V_{marked} = \emptyset$.

For the graph instance model $G_1$ of Example 3, $m^{(2)}$ is established in Example 7 as one of the matches of the LHS pattern $marked$, and the effect of applying the rule on this match would be $G_1.ApplyRule(removeToken,m^{(2)}).\delta = \langle \{k_2 :: Token\}, \{p_1 \xrightarrow{(m_2 :: Marking)} k_2\}, \emptyset, \emptyset \rangle$.

The attributed version of the rule $removeToken$, in context of the attributed metamodel of Example 4, is shown in Figure 2.17. The LHS matches a marked place $P$ and finds its Marking value assignment $MX$, which points to non-zero Integer $X$, which is - as expressed by a Successor data relation - one more than Integer $Y$. The RHS pattern is similar to the LHS, but contains $MY$ instead of $MX$, which points to $Y$ instead of $X$. Since $V_{LHS} \setminus V_{RHS} = \{MX\}$ and $V_{RHS} \setminus V_{LHS} = \{MY\}$, applying the rule will remove the $MX$ value assignment, and insert $MY$ in its place, in effect decreasing the marking of $P$ by one.
CHAPTER 2. BACKGROUND

Figure 2.16: GT rule for removing a token

Figure 2.17: GT rule for removing a token (attributed version)
Chapter 3

Incremental Graph Pattern Matching

3.1 Incremental graph pattern matching basics

Incremental pattern matching is an approach that has been shown by numerous studies over the years (e.g. [VV04, VVS06, GJR10][16]) to exert favorable performance characteristics in many practical use cases. The core idea is to improve the pattern matching phase by storing (caching) precomputed results and maintaining them as the model changes.

In the following sections, I introduce a formal treatment of incrementality and related concepts in the context of graph pattern pattern matching. The language introduced here will be used to review various incremental approaches and describe the proposed solution.

3.1.1 Stateful pattern matching

The following paragraphs introduce a formalization of pattern matchers that take advantage of incrementally maintained cache storages. The previously introduced formal description has to be extended by the notion of the stateful pattern matcher \( PM \). A stateful pattern matcher has an internal state \( s \), with \( s \in S \) for a set \( S \) of valid states (state space). The matcher internal state can preserve results of calculations between pattern matcher invocations; it may be responsible e.g. for storing the match sets of patterns. This means that the configuration of the evolving model can only be described by both the graph model and the matcher internal state.

**Definition 39 (Configuration of evolving graph with stateful pattern matching)** In case of stateful pattern matching with state space \( S \) on a graph model over metamodel \( MM \), the current configuration of the evolving graph is a pair \( \langle G, s \rangle \in \text{Graphs}^S_{MM} \) that consists of the current graph model \( G \) and the matcher internal state \( s \), where the set of valid configurations \( \text{Graphs}^S_{MM} \subset \text{Graphs}_{MM} \times S \) is characteristic to the stateful pattern matcher.

**Definition 40 (Operations with stateful pattern matching)** A model access operation \( Op \) in case of stateful pattern matching is a partial function \( Op : \text{Graphs}^S_{MM} \to (\text{Graphs}^S_{MM} \times \text{Out}_{Op}) \) that maps a configuration \( \langle G, s \rangle \) to the tuple \( \langle G', s', \text{out} \rangle \) with resulting configuration \( \langle G', s' \rangle \in \text{Graphs}^S_{MM} \) and output \( \text{out} \in \text{Out}_{Op} \). The application of the operation on configuration \( \langle G, s \rangle \) is denoted as \( \langle G, s \rangle.OP \), while \( \langle G, s \rangle.OP.r \) denotes the resulting configuration and \( \langle G, s \rangle.OP.out \) denotes the yielded output. All operations conforming to Definition 16, are also considered compatible with the updated definition, with \( \langle G, s \rangle.OP = \langle G.OP.r, s, G.OP.out \rangle \).
The last statement essentially means that all queries and model manipulation operations defined so far are also able to act on the composite configuration, but they do not directly use or change the internal storage of the stateful matcher (though manipulative operations will result in subsequent maintenance of the internal state).

Now the stateful pattern matcher can be defined as follows. Since in a stateful context, pattern matching is a model access operation that is applied on configuration \( \langle G, s \rangle \), the stateful matcher can rely on its internal state \( s \) when evaluating queries, in addition to the graph model \( G \). For example, the match operation could be trivial if the match set is stored in its entirety within \( s \). A stateful pattern matcher may even update its internal state after a query operation, e.g. to store new results computed during query evaluation for later reuse. Finally, a stateful pattern matcher additionally provides a function that updates its state \( s \) to \( s' \) as a reaction to a model manipulation operation causing change \( \delta \) in the graph model \( G \); the difference between \( s' \) and \( s \) is expected to reflect the effects of \( \delta \).

**Definition 41 (Stateful pattern matcher)** A stateful matcher is a structure \( PM = \langle S, s_0, \text{Match}, \text{Maintain} \rangle \), where \( S \) is the set of valid internal states, \( s_0 \in S \) is the matcher null state, \( \text{Match} : \text{Patterns}_{MM} \to \text{Graphs}_{SM}^S \to (\text{Graphs}_{SM}^S \times \text{Out}_{\text{Match}}) \) is a query operation parameterized by a pattern, and \( \text{Maintain} : \text{Deltas}_{MM} \to \text{Graphs}_{SM}^S \to (\text{Graphs}_{SM}^S) \to \text{Out}_{\text{Maintain}} \) is an operation parameterized by a graph delta:

- For a given pattern \( P \in \text{Patterns}_{MM} \), the pattern matching query \( \text{Match}(P) \) computes the result \( \langle G, s \rangle.\text{Match}(P).out = \text{MatchSet}_G^P \) of this pattern matching operation; it also computes a new internal state \( s' \in S \) for the matcher as \( \langle G, s \rangle.\text{Match}(P).r = \langle G, s' \rangle \), with the graph model itself unchanged (hence it is a query).

- For a given \( \delta \in \text{Deltas}_{MM} \), the matcher maintenance routine \( \text{Maintain}(\delta) \) yields no useful output (\( \text{Out}_{\text{Maintain}} \) contains a single element), but results in a configuration \( \langle G, s \rangle.\text{Maintain}(\delta).r = \langle G', s' \rangle \) composed of a new internal state \( s' \in S \) of the PM valid for the new graph model \( G' = G + \delta \).

- The null state \( s_0 \) is the default internal state associated with an empty graph model, so that \( \langle \emptyset, s_0 \rangle \) is considered a valid configuration.

For an evolving graph model, the net effect of manipulation operations between two PM queries will be fed into \( \text{Maintain} \) to update the matcher internal state, as shown in the next definition.

**Definition 42 (Execution trace of evolving graph with stateful pattern matcher)** The execution trace of an evolving graph is the sequence \( \langle G_1, s_1 \rangle \xrightarrow{\text{step}_1} \langle G_2, s_2 \rangle \xrightarrow{\text{step}_2} \langle G_3, s_3 \rangle, \ldots, \langle G_n, s_n \rangle \) of configurations and trace steps inbetween. Each configuration \( \langle G_i, s_i \rangle \in \text{Graphs}_{SM}^S \) is a valid configuration with respect to the same metamodel \( MM \) and same matcher state space \( S \). Each trace step \( \text{step}_i \) is associated with a transaction operation \( Op_i \), which is either a pattern matching operation \( \text{Match}(P_i) \), or a model manipulation operation \( \text{Mod}_i \). Each subsequent configuration \( \langle G_i, s_i \rangle \) along the trace is obtained from the previous by applying the transaction operation and if needed, maintaining the pattern matcher internal state to reflect the changes; i.e. \( \langle G_i, s_i \rangle.\text{Match}(P_i).r = \langle G_{i+1}, s_{i+1} \rangle \) with \( G_{i+1} = G_i \) in case of a PM query; and \( \langle G_i, s_i \rangle.\text{Maintain}((\langle G_i, s_i \rangle.\text{Mod}_i, \delta)).r = \langle G_{i+1}, s_{i+1} \rangle \) in case of a manipulative transaction.

As a special case of applying \( \text{Maintain} \), the stateful matcher can be initialized on any graph model \( G \) as \( \langle G, s \rangle = \langle \emptyset, s_0 \rangle.\text{Maintain}(G - \emptyset).r \). This formula means that the current configuration
is obtained from the null configuration (empty graph model and matcher in null state) as if the delta between the current graph model and an empty graph model has just been added to the empty graph model.

### 3.1.2 Algorithmic complexity of stateful pattern matching

For graph model $G$, let $|G|$ denote the number of graph elements excluding the data algebra. The memory complexity of an implementation system containing a graph model and any pattern matching mechanism is at least $\Omega(|G|)$, since the model elements themselves have to be stored even if $s$ is empty.

Querying $\langle G, s \rangle$.Match$(P)$, more precisely enumerating the match set $\text{MatchSet}_G^P$ of a pattern $P$ has a time complexity of at least $\Omega(|\text{MatchSet}_G^P|)$, as each match has to be enumerated. In case of seeded pattern matching, this lower bound is $\Omega(|\text{ResultSet}_G^P(\mu)|)$; keep in mind that the result set can be much smaller than the entire match set.

For an implementation system that performs stateful pattern matching on a evolving graph, the internal state of a stateful matcher comes at the cost of additional memory consumption, as well as additional time required for maintenance (total runtime of the maintenance routine). The reason of employing stateful matching is that the execution of pattern matching $\langle G, s \rangle$.Match$(P)$ can be much more efficient than the stateless $G$.Match$(P)$, if the contents of $s$ is chosen well. In particular, if $s$ contains the match set, this query operation is virtually for free (more precisely proportional in execution time to the result set, due to the need to enumerate). This case is formalized below.

**Definition 43 (Fully caching pattern matcher)** The following notions of matchers fully caching patterns are introduced:

- A stateful pattern matcher is fully caching for pattern $P$ in configuration $\langle G, s \rangle$ iff the internal storage $s$ contains the match set $\text{MatchSet}_G^P$.

- A stateful pattern matcher is practically fully caching for pattern $P$ in configuration $\langle G, s \rangle$ iff the match set $\text{MatchSet}_G^P$ is computable from $\langle G, s \rangle$ efficiently, in $O(|\text{MatchSet}_G^P|)$ time\(^1\).

- A stateful pattern matcher is (practically) fully caching for pattern $P$ if it is (practically) fully caching for $P$ in all configurations of the trace.

- A stateful pattern matcher is on-demand (practically) fully caching for a pattern $P$ that is matched at least once iff the matcher is (practically) fully caching for $P$ in the configuration after the first application of Query$(P)$ and in all subsequent configurations.

Let $\text{Patterns}_{\text{Required}}$ denote the set of patterns that the fully caching stateful pattern matcher is required to cache. Essentially, in a fully caching pattern matcher the matches of these patterns are stored explicitly in $s$. All matches of a graph pattern $P \in \text{Patterns}_{\text{Required}}$ can be retrieved in time proportional to the result set by eliminating the need for recomputing existing matches.

In exchange, memory consumption is increased by at least $\Omega(\sum_{P \in \text{Patterns}_{\text{Required}}}|\text{MatchSet}_G^P|)$, i.e. total memory complexity (including the graph model) becomes $\Omega(|G| + \sum_{P \in \text{Patterns}_{\text{Required}}}|\text{MatchSet}_G^P|)$. For some techniques, e.g. if they store in $s$ the matches only, this lower bound is strict; in their case the size of the internal storage can grow at most linearly with the match set (plus the model), permitting only inexpensive but important helper structures, such as indexes, or continuation structures of lazy evaluation (e.g. LEAPS in Section 3.2.2). This memory-efficient property of certain fully caching matchers is formalized below.

\(^1\) $O(|\text{ResultSet}_G^P(\mu)|)$ in case of seeded pattern matching
**Definition 44 (Minimal cache)** A fully caching pattern matcher for $P$ has minimal cache iff $|s| = O(|G| + \sum_{P \in \text{Patterns}_{\text{required}}} |\text{MatchSet}^P_G|)$ in each configuration $(G, s)$. Otherwise, the matcher is non-minimal; it maintains additional caches for storing auxiliary results (e.g. partial matches).

As a further drawback of fully cached pattern matchers, there is an overhead of updating the cache as the match sets change. This is either done entirely in the maintenance routine $\text{Maintain}$ directly when the model manipulation happens, or partially deferred until the next query operation. In either way, a certain amount of computation is needed to have the caches refreshed by the time the next query is issued. For simplifying the discussion, we will assume eager maintenance, i.e. all such cache maintenance happens in $(G, s).\text{Maintain}(\delta)$ so that any subsequent $\text{Match}(P)$ does not have to modify $s$ (provided the PM is already fully caching $P$). In essence, the burden of cache maintenance is carried by model manipulation transactions in case of eager maintenance. This simplification will make complexity analysis easier, although in some cases the alternative of partially deferring the maintenance (lazy maintenance) can have a certain impact on performance. These effects may include a reduced maintenance overhead when some changes are undone by subsequent manipulations before the next query, as well as increased memory consumption by unnecessarily preserving a log of deleted elements (and corresponding pattern matches) for potentially a long time.

As the most central concept to this thesis, incrementality in stateful pattern matchers is the aim of reducing the maintenance cost in $(G, s).\text{Maintain}(\delta)$ by reusing (or rather building upon) existing results in $s$.

**Definition 45 (Incremental pattern matching)** A stateful pattern matcher is incremental if $(G, s).\text{Maintain}(\delta)$ computes the difference $s' - s$ from $s$ and $\delta$ rather than recomputing $s'$ from scratch. A pattern matcher is fully incremental if it is fully cached and incremental.

Clearly, the maintenance cost of incremental pattern matching is associated with the change from $s$ to $s'$. Including the processing of the model delta, this takes at least $\Omega(|\delta| + |s' - s|)$ time, giving a lower bound for maintenance complexity. For some techniques, this bound is strict:

**Definition 46 (Swift incremental pattern matching)** An incremental pattern matcher has swift updates if $(G, s).\text{Maintain}(\delta)$ has at most $O(|\delta| + |s' - s|)$ complexity.

It is theoretically possible that a fully incremental matcher with swift updates has a more costly maintenance routine than a non-swift fully incremental counterpart, if the latter has asymptotically smaller cache. However, trade-offs between cache size and maintenance time are more usual, see e.g. the discussion of Rete vs. TREAT in Section 3.2.2 and Section 3.3.4.

As $s$ contains the match set in case of a fully cached matcher, the amount of change in the match set can have a significant influence on the performance of a fully incremental matcher. Since the change of match sets is such a central concept in analyzing the efficiency of incremental pattern matching, the concept of delta match set is formally introduced below, in analogy with the notion of graph delta.

**Definition 47 (Match set delta)** If graph delta $\delta$ is applied to model $G$, the delta match set of a pattern $P$ in context of $G \oplus \delta$ (where $G \oplus \delta$ is a notational shorthand for $(G, \delta)$ the context of change) is the following: $\Delta \text{MatchSet}^P_{G\oplus\delta} := \langle \delta_-, \delta_+ \rangle$ where $\delta_- = \{-m \mid m \in \text{MatchSet}^P_G \setminus \text{MatchSet}^P_{G+\delta} \}$ and $\delta_+ = \{ +m \mid m \in \text{MatchSet}^P_{G+\delta} \setminus \text{MatchSet}^P_G \}$. 


3.2. RELATED WORK

Due to the busy nature of this notation, the specification of change context $G \oplus \delta$ can be omitted when clear, resulting in the notation $\Delta \text{MatchSet}^P$. Deltas of result sets of seeded pattern matching can be defined analogously.

In case of fully cached matchers, the change of the cache contains at least the change of the cached match sets, so $|s' - s| = \Omega(\sum_{P \in \text{Patterns}\text{Required}} |\Delta \text{MatchSet}^P|)$, making the time complexity of maintenance at least $\Omega(|\delta| + \sum_{P \in \text{Patterns}\text{Required}} |\Delta \text{MatchSet}^P|)$.

The latter lower bound is not strict; unfortunately, no fully cached pattern matcher can deliver $\Theta(|\delta| + \sum_{P \in \text{Patterns}\text{Required}} |\Delta \text{MatchSet}^P|)$ maintenance complexity.

Theorem 1 It is not possible for a fully cached matcher to have a maintenance time complexity of $O(|\delta| + \sum_{P \in \text{Patterns}\text{Required}} |\Delta \text{MatchSet}^P|)$ for any arbitrary set of patterns $\text{Patterns}\text{Required}$.

Proof If one grows a graph model $G'$ from scratch (i.e. $G = \emptyset$ and the context is $\emptyset \oplus \delta$), such a very efficient fully cached matcher would compute the matches of the graph patterns in linear time. That would contradict the theorem [Grö92] that subgraph isomorphism (a sub-problem of PM) is at least of $\Omega(\sqrt[3]{|G'|})$ complexity.

Theorem 2 It is not possible for a fully incremental matcher to have both minimal cache and swift updates.

Proof In case of minimal cache, $|s' - s_0| = O(|s'|) = O(|G'| + \sum_{P \in \text{Patterns}\text{Required}} |\text{MatchSet}^P_{G'}|)$. Due to swiftness, $(\emptyset, s_0).\text{Maintain}(G' - \emptyset)$ has a time complexity of $O(|G' - \emptyset| + |s' - s_0|) = O(|G'| + \sum_{P \in \text{Patterns}\text{Required}} |\text{MatchSet}^P_{G'}|)$, contradicting [Grö92] once again.

In workloads where large models are affected by manipulative transactions with a relatively moderate delta, the match sets typically also experience only relatively minor change ($\Delta \text{MatchSet}^P$ will be small), and thus the updated internal state $s'$ is expected not to differ greatly from $s$. In these cases, significant runtime performance benefits are expected from incrementality.

3.2 Related work

Incremental updating strategies have been widely used in different fields of computer science. Each of the following sections reviews incremental techniques in a given field, assessing them in light of Section 3.1. Some of the properties of these algorithms are summarized by Table 3.1.

3.2.1 Related work: incremental graph pattern matching in graph transformation

Now we give a brief overview on incremental techniques that are used in the context of graph transformation.

Attribute updates. The PROGRES [Sch90] graph transformation tool supports an incremental technique called attribute updates [Hud87]. At compile-time, an evaluation order of pattern variables is fixed by a dependency graph. At run-time, a bit vector is maintained in $s$ for each model entity expressing whether it can be bound to the variables of the pattern. When model entities are deleted, some validity bits are set to false by the maintenance routine, which might invalidate partial matches immediately. On the other hand, new partial matches are only lazily computed during Query, taking advantage of the bit vectors, therefore this strategy is not fully caching.

Notification arrays. [VVS06] proposes a fully incremental graph pattern matching technique, which constructs and stores in $s$ a tree (essentially a search tree) for partial and complete matches of
CHAPTER 3. INCREMENTAL GRAPH PATTERN MATCHING

<table>
<thead>
<tr>
<th>Method</th>
<th>Paper</th>
<th>Fully caching</th>
<th>Minimal cache</th>
<th>Swift</th>
<th>Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRES</td>
<td>[Hud87]</td>
<td>-</td>
<td>N/A</td>
<td>N/A</td>
<td>-</td>
</tr>
<tr>
<td>Notification arrays</td>
<td>[VVS06]</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Incremental SLD</td>
<td>[HLR06]</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Rete</td>
<td>[For82]</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>TREAT</td>
<td>[ML91]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Rete*</td>
<td>[WM03]</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Gator</td>
<td>[HH93]</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LEAPS</td>
<td>[Bat94]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RDB helper tables</td>
<td>[VV04]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Counting</td>
<td>[GMS93]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DRed</td>
<td>[GMS93]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Chain query FOIES</td>
<td>[DDS+93]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>OCL (Cabot et al.)</td>
<td>[CT09]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OCL Impact Analyzer</td>
<td>[UGH11]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Groher-Reder-Egyed</td>
<td>[GRE10]</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Praxis</td>
<td>[FBB+12]</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 3.1: Reviewed incremental evaluation approaches

A pattern as nodes. The tree is incrementally updated in the maintenance routine when the model changes. The complete matches are available to Query as lowest-level leaves of the tree. As a novelty, notification arrays are introduced for speeding up the identification of such partial matches that should be incrementally modified by Maintain. The main advantage of this solution is that each node of the tree contains very little information (a single variable assignment, which is shared by the entire subtree of the node), which possibly saves a significant amount of memory by reducing the size of s. Still, the cache is non-minimal, as s also includes partial matches that cannot be extended to valid matches of the pattern, and in some cases these superfluous partial matches may dominate legitimate matches in memory consumption.

**Incremental SLD resolution.** The fully incremental approach of [HLR06] performs pattern matching by building and incrementally maintaining Prolog-like SLD resolution trees in s that evaluate patterns formulated as logical predicates. This resolution tree, unlike a simple search tree, can be incrementally maintained not only upon model modification, but also when patterns themselves are modified. Such uniform, incremental handling of model elements and patterns can be considered a unique, advanced feature of the approach. However, it is also expected to consume significantly more memory than the approach in [VVS06] (and it is similarly non-minimal).

**Production rule techniques.** Bunke et al. [BGT91] adapted the Rete algorithm originally developed in the domain of production rule systems (see the following section for details) for pattern matching in a GT engine. The presented solution supported a simple core graph pattern language, without any of the advanced language elements discussed in Section 2.2. Performance evaluation was limited to small synthetic example models up to 80 elements, therefore its conclusions do not translate directly to industrial models that can be several orders of magnitude larger and run on modern hardware. My work extends this approach by providing support for all pattern language constructs of Section 2.2, as well as providing measurements on a wide range of benchmarks, including industrial models.
3.2. RELATED WORK

3.2.2 Related work: incremental matcher algorithms for production rule systems

Outside the context of GT and graph pattern matching, production rule engines (including rule-based expert systems such as OPS5 [BFKM85] or business rule systems such as Drools [P^+]) apply various incremental techniques for finding and maintaining the conflict set, i.e. the set of fact tuples that satisfy preconditions of production rules. Such a precondition can be analogous to an LHS graph pattern, where the conflict set would correspond to the match set.

Rete networks [For82] is a well-known algorithm stemming from rule-based expert systems. Rete has already been used as an incremental pattern matching technique in several application scenarios including the recognition of structures in images [BGT91], and the co-operative guidance of multiple uninhabited aerial vehicles in assistant systems as suggested by [MMS07]. Rete stores in its internal state the matches of the patterns (making Query instantaneous), as well as the match sets of certain partial patterns (used to speed up the maintenance routine); this makes Rete fully incremental, swift and non-minimal. See Section 3.3 for a more detailed introduction and formalization of Rete. Being a flexible and easy-to-implement technique with swift updates, Rete was chosen to be used throughout my thesis. Extending the original application [BGT91] of Rete for GT, it is adapted as a graph pattern matcher for a more expressive graph pattern language in Section 3.4 and my publication [23].

TREAT [Mir87, ML91] aims at minimizing memory usage while retaining the incremental property of pattern matching and instant accessibility of the entire conflict set. Only the match sets are stored in s, no partial matches are memoized in the matcher state; thus TREAT is fully incremental, non-swift and has minimal cache. Advantages include a reduced memory consumption, and (in case of restricted pattern languages) a simpler maintenance routine for deletion. The performance of the maintenance routine for insertion, however, may be negatively impacted by the unavailability of cached partial matches. Some sources claim [Mir87] that TREAT is faster than Rete (the difference is in constant coefficients, not in terms of complexity class), others disagree ([NGR88] states various arguments and measurements in favor of Rete). Note, however, that this performance assessment has been carried out for rule-based expert systems with usage workloads significantly different from the MDE context (see more explanation at LEAPS).

Rete* [WM03] is a common generalization of Rete and TREAT that attempts to strike a balance between memory size and performance by storing in s and maintaining partial matches only for the most frequently needed partial patterns; the two extreme cases for this memory retention policy are TREAT and Rete. Gator networks [HH93] are a similar hybrid approach, where a TREAT-like algorithm can be embedded into a multi-input node participating in the Rete net. Both of these approaches are therefore fully incremental and non-minimal (unless they degenerate into TREAT).

The LEAPS algorithm [Bat94] is a fully incremental approach with minimal cache; similarly to TREAT, no partial matches are stored in s, only the match sets (and potentially continuations for lazy evaluation). The approach can be characterized by lazy evaluation to avoid manifesting tuples unnecessarily, by depth-first firing, and by the introduction of timestamps to be able to reconstruct earlier conditions (‘time travel’) for the lazy evaluation. LEAPS is claimed to be substantially better than either Rete or TREAT at both time and space complexity. However, these investigations are aimed at production rule systems, where the conflict set (match set) is empty during steady state, and thus rule firing order and temporary storage used in the maintenance routine have a significant impact on performance. Most of the optimizations in LEAPS are therefore ineffective in other applications of pattern matching, including many use cases of graph pattern matching where uncontrollable depth-first rule firing is not acceptable. However, a LEAPS-based trigger engine (see Section 7.5.3.2 for a discussion of event-driven rule execution) is still worth considering in the future.
3.2.3 Related work: incremental maintenance in databases

Relational databases. Industrially wide-spread and among the most successful products of software engineering, _relational databases_ (RDBs) manage large quantities of information structured according to the well-known _relational data model_ (schema, table/relation, row/tuple, column, key). Conventional RDBs store all data on disk (using memory as cache), while others are specifically designed to operate entirely _in-memory_ for better performance. The most common interface language for RDBs by far is _SQL_, capable of schema definition, data manipulation and querying alike.

SQL dialects typically support code reuse to help programmers and also to take advantage of performance benefits associated with e.g. pre-compiling and pre-optimization. A reusable data manipulation program segment is called a _stored procedure_, while a reusable query expression defines a _view_. A _trigger_ on a table is a special stored procedure that is invoked upon each row manipulation affecting the table (receiving the description of the change as input), thereby facilitating event-driven programming. The technological advantage of an RDB-based implementation would be the compatibility with existing models stored in off-the-shelf RDB products, preferably not requiring the modification of any legacy programs that already exist and manipulate the model.

In the context of relational databases, the cached result of a query is called _materialized view_, although it is not assumed by default to be up-to-date. Some commercial database engines provide this feature along with the option of automatic and incremental maintenance (i.e. automatically calling the maintenance routine as described in Section 3.1.1). In this case, reads from the materialized view are a drop-in replacement for the original query, making the system fully incremental with minimal cache. However, in mainstream databases this non-standard feature is typically restricted to a subset of SQL queries which is insufficient to express complex graph patterns (especially NACs); therefore it was not possible for this thesis to rely on it. For example, Flexviews for MySQL and Indexed Views in MS SQL do not support outer joins (or existence checking) that is required for efficient NAC enforcement, while Oracle’s Materialized Views do not even support top-level inner joins, and finally there is no built-in incremental maintenance at all in PostGreSQL.

The lack of built-in support can be compensated by defining helper tables (instead of materialized views) as the internal state of the matcher and maintaining them programmatically via an SQL maintenance routine. Like the materialized views solution, this strategy is also fully incremental with minimal cache. The paper [VV04] is one of the first to suggest the idea of incremental LHS evaluation in graph transformation, and describes a proof-of-concept experiment implemented in a RDB, following this approach. Since the main focus of the paper was the feasibility of incremental pattern matching in general, the particularities of the RDB-based implementation and the automated mapping were not elaborated in detail. Furthermore, the consistency of the incremental cache is only guaranteed if the graph model is restricted to evolve along the specified GT rules only: external programs manipulating the underlying database may cause inconsistency of the match results. This is a consequence of making the incremental maintenance of the results (the invocation of Maintain) an explicit part of the manipulation phase of GT rule execution, which is not invoked when pre-existing programs manipulate the model, as opposed to the reliable built-in mechanism envisioned in Section 3.1.1. I have addressed these drawbacks in Section 4.3 and my paper [7] by a significant conceptual extension over [VV04]. The advanced solution features trigger-based automated maintenance of pattern match sets according to LEAPS (see in Section 3.2.2), even upon model manipulations carried out by unmodified legacy programs.

The incremental matching algorithm Rete (already discussed in Section 3.2.2) is integrated into an RDB in [JCH05], but the user formulates queries in SQL as opposed to declarative graph patterns; also, Rete maintenance is performed periodically, not by event-driven triggers, which is an additional
drawback.

**Deductive databases.** There are also significantly more powerful approaches [GMS93, DDS+93] for fully incremental query evaluation over logic databases, that support the highly expressive Datalog language. The greatest challenge in terms of correctness, performance, and implementation difficulty is the handling of recursive Datalog queries (especially when combined with negation), thus, algorithms in this field sacrifice performance, expressiveness (in case of [DDS+93]) and simplicity to address this issue\(^2\), which it is not relevant in the context of non-recursive graph patterns. Even though there exists a recursive extension to the language of graph patterns (see Section 2.2.2.3 for detailed discussion), the primary focus of the current thesis is on conventional graph patterns, therefore the benefits and challenges of recursive queries are out of scope. Since unlike SQL, Datalog is not supported by mainstream commercial databases, it would also fail to provide the expected benefit of industrial compatibility.

### 3.2.4 Related work: incremental maintenance of queries in MDE

OCL [OMG12a] is a well-known model query language with powerful features. The language is very expressive, surpassing the power of first order logic by constructs such as collection aggregation operations (sum, etc.) and ordered collections. It is important to note that the latter is not possible to express using graph patterns.

Due to the expressive power of such OCL constructs, the Rete-based approach that will be discussed in this thesis is not applicable for queries formulated as OCL expressions. It is possible, though, to identify sublanguages of OCL that can be translated to graph patterns [WTEK08], so that graph pattern-based approaches, including the results of this thesis, can be applied.

There are, however, several alternative approaches that provide incremental evaluation of OCL queries. Cabot’s approach [CT09] derives an (over-estimating) re-evaluation action specific for each query and each elementary model change, analogously to TREAT. The Impact Analyzer [UGH11] extension of MDT-OCL [Ecl11] relies on static analysis of OCL expressions when computing an over-estimate of queries that need to be re-evaluated. The Groher-Reder-Egyed approach [GRE10] for incremental constraint checking is independent from the constraint language, but can be instantiated for OCL. The strategy is to wrap the model into a model access layer that records elementary model query operations during the constraint checking (OCL query evaluation); later the query can be re-evaluated if any of the recorded elementary queries are affected by a change.

Departing from OCL, Praxis [FBB+12] uses a rule-reduction technique to provide minimal cache incremental instance model validation. It represents the model and deltas as Prolog facts using an operation based approach. Constraints are represented as Prolog rules, which are evaluated when they are triggered based on the user-provided impact list.

Case study-driven comparative performance benchmarking of incremental model query evaluation technologies is a currently ongoing effort.

### 3.3 Principles of the Rete algorithm

The incremental evaluation algorithm Rete [For82] has first been applied to the problem of graph pattern matching in [BGT91]. In this section, I first present a basic high-level overview of the basic concept of Rete, and then provide a novel formalization consistent with the notion of stateless

\(^{2}\)With the exception of the simple Counting algorithm [GMS93] (largely analogous to TREAT discussed in Section 3.2.2), which is quite fast (also confirmed by our limited experiments), but incompatible with recursion.
pattern matcher defined in Section 3.1. Section 3.4 will use this formalization to greatly extend the results of [BGT91] and support the advanced features of the graph pattern language introduced in Section 2.2.1.

3.3.1 High-level overview of components and structure

The Rete network, originally designed [For82] for rule-based expert systems, is a DAG (directed acyclic graph). Each Rete node in the network is associated with a pattern, which may be one of the Patterns Required patterns, or a subpattern of a Patterns Required pattern for internal use. The node contains the match set of the associated (sub)pattern; the elements of the match set are represented as tuples. The edges of the Rete net are used to propagate changes in such match sets. Although in practice, there are several variations of the basic idea, an overview of commonly encountered features of Rete can be given here.

Rete networks contain nodes of various types. There is a distinguished set of nodes (sometimes one single node) called input nodes that contain the asserted facts of the knowledge base (i.e. the graph model). The other nodes are operation nodes. The alpha operation nodes are connected with an edge to a parent node (usually the input node or another alpha node); they filter the contents of the parent node according to some constant criterion (e.g. type, attribute range). Arguably the key components of Rete are the beta operation nodes, that have two separate input slots, each connected to a node in the network. The contents of a beta node are tuples built as some kind of composite of two input tuples (one from each slot) that are paired by some criteria. Typically, beta nodes perform a natural join operation (as in relational algebra) on the contents of their parent nodes. Finally, a distinguished production node for each pattern collects the matches of the pattern. This simplified structural overview of Rete is illustrated by Figure 3.1.

The Rete matcher is highly flexible, making a wide range of pattern matching strategies possible. A single node may have any number of children. This enables nodes to be shared between patterns or between parts of the same pattern. A union node can have several incoming edges and treat the union of the contents of its parents as its input. A pattern can be matched by a linear sequence of beta nodes, each expanding the partial match by an additional fact, or a more complex (but less deep) network composed of converging subnetworks responsible for different parts of the pattern.

---

If recursive patterns are involved, the DAG property may be violated

Some versions of the algorithm require the secondary input to be a child of an alpha node
3.3. PRINCIPLES OF THE RETE ALGORITHM

Figure 3.2: Basic graph pattern capturing inhibition

Note that while this high-level overview refers to tuples as actually being stored at nodes, this is merely a way of explaining the basics of the Rete concept. In actual implementations this may not be the case, as it is possible for some nodes not to contain a memory. Some Rete network descriptions put emphasis on isolating local memory storages, that are components responsible for storing (and possibly indexing) tuples, and all memories together form a distributed working memory. It is possible to distinguish between alpha memories and beta memories, based on whether they store tuples that are simple asserted facts or compound tuples output by beta nodes. Section 3.4 covers the memory aspects of my implementation in detail.

Example 18 The following example relies on the inhibits pattern, which expresses that a marked place inhibits a transition via an inhibitor arc. The pattern was used before in Example 15 on page 32; it is redefined here using the basic formalism (i.e. without pattern composition) and depicted by Figure 3.2. A Rete net that matches this pattern is depicted in Figure 3.3, where each Rete node is shown as a grey box, interconnected by grey arrows pointing from parent node to child node. The Rete net is composed of input nodes r₁ and r₂, and a beta operation node r₃. The role of the input node r₁ is to cache all relations of type Marking, while InhibitorArc relations are stored at r₂. From these two parent nodes, r₃ derives the set of Petri places that are marked and are inhibiting a transition, by applying the relational algebra operation of natural join. The result computed at this operation node is, by definition, the match set of the inhibits pattern, making r₃ a production node that is fully caching inhibits.

If this Rete net is initialized on the instance model of Example 3 on page 13, nodes will contain in their memories the matches of their associated patterns. This is shown in Figure 3.4, with individual matches represented as white boxes inside the node that contains them (ignore the red and green boxes for now). Input node r₁ will contain one tuple for each of the Marking relations m₁, m₂, m₃ and m₄. Input node r₂ will cache the single InhibitorArc relation g₁. Finally, join node r₃ will combine Markings with incident InhibitorArcs, and thereby compute the relational join of these match sets, which now happens to be the empty set. This means that the pattern inhibits has no matches, i.e. there is no inhibited transition in the model.

3.3.2 High-level overview of operation

Once the Rete net is built, finding the matches of a pattern is as simple as retrieving the contents of the production node corresponding to the pattern.

If the graph model undergoes changes, the Rete network has to be updated in order to keep the match sets up-to-date. Whenever a new graph element is inserted, a positive update token containing the new fact is passed to the input node. This will start a propagation of change information through the Rete net. Nodes receive change tokens on their input, update their memories accordingly, and
propagate the changes of their associated match sets as update tokens on their outgoing edges. For instance, alpha nodes in filtering roles will pass a token to their children if the fact enclosed in the token satisfies the filter condition associated with the alpha node. Upon receiving an update token on one of their input slots, beta nodes for natural join will look up tuples from their other input slot that are compatible with the incoming token according to the join rules; for each suitable pair that is found, a new composite tuple is created from them and propagated to the children of the beta node.

If an element is removed from the model, the Rete has to be updated as well. Analogously to the previous procedure, negative update tokens are propagated in the network in this case. The only key difference is that nodes have to “invert” their operation; e.g. tuples have to be removed from the memories instead of being added.
3.3. PRINCIPLES OF THE RETE ALGORITHM

Example 19 Continuing from Example 18, let us now observe the consequences of executing the manipulative transaction $Fire(t_2)$ (see Example 6 on page 21). The contents of the Rete net will be maintained according to the graph delta; first the changes are performed in the input nodes (directly reflecting the graph delta), and then they propagate down from parent node to child node, until there is nothing left to update.

Figure 3.4 shows matches in the negative delta of a node as red boxes, and matches in the positive delta as green boxes. In this case, the delta of $r_1$ will remove one match ($Marking m_4$) and add one match ($Marking m_5$), while the match set of $r_2$ is unchanged. Then, the join node $r_3$ will incrementally compute its own delta, by examining each incoming change and consulting the other parent node to determine how it affects the outcome. As no matches stored in $r_3$ were derived from $m_4$, none will have to be removed after $m_4$ is deleted. When $m_5$ is inserted, however, it can be joined with $InhibitorArc g_1$ from the other parent, as both are connected to the same Place $p_2$.

Therefore $r_3$ registers one new match in the cached match set of pattern inhibits, that identifies $p_2$ as a marked place inhibiting $t_1$. Were there any child nodes of $r_3$, they would have to be updated according to this delta.

If the Rete net is queried for the matches of pattern inhibits, the query operation consist of enumerating the matches that are stored directly at $r_3$. In this case, there are no matches before the update, and the single match with $p_2$ after the update.

3.3.3 Formalization

In the following, a simplified formal treatment of the structure and operation of a Rete-based incremental pattern matcher is given, on a very high level of abstraction. Some details are chosen in a way to make the formalization as simple as possible, not necessarily reflecting the details of an implementation. Individual computation operations are not discussed, neither are implementation questions and performance improvement techniques. These will be discussed in Section 3.4, along with the adaptation to a pattern language / modeling platform, and the procedure of constructing a Rete net to match a given pattern.

If Rete is utilized to fully cache a pattern $P$, there will be a Rete node that contains the match set of $P$. Due to the way Rete works, there need to be additional Rete nodes that cache subpatterns of $P$, which contain a subset of variables and constraints. Each node has a dedicated “storage area” in the internal state of the matcher, where it can store its own internal node memory including (but not necessarily restricted to) the match set of its associated pattern. There are two main kinds of Rete nodes: input nodes and operation nodes.

Operation nodes (including alpha and beta nodes) compute and incrementally maintain their contents based on the contents of a set of parent nodes. An operation node defines this computation through the initial “null state” of its internal memory (when all parent nodes have empty match sets) and a node maintenance routine that updates the node internal memory based on the deltas (and original contents) of parent nodes.

Definition 48 (Rete operation node) A Rete operation node is a structure $r = (P_r, Parents_r, M^0_r, Maintain_r)$ that is associated with:

- a subpattern $P_r \subseteq P$ where $P$ is a pattern cached by the pattern matcher
- a tuple of parent nodes $Parents_r = \langle r^1, r^2, \ldots, r^k \rangle$
- a null state of the node memory $M^0_r$ that should be valid in case all parent nodes have an empty match set
• a node maintenance routine \( \text{Maintain}_r \), which can \( \text{incrementally} \) calculate the new internal memory \( M'_r \) of the node from the current internal memory \( M_r \), when parameterized for each parent node by the current match set of the associated pattern of the parent and the corresponding match set delta: \( M'_r = M_r \cdot \text{Maintain}(\text{MatchSet}^{P_1} \oplus \Delta \text{MatchSet}^{P_1}, \text{MatchSet}^{P_2} \oplus \Delta \text{MatchSet}^{P_2}, \ldots, \text{MatchSet}^{P_k} \oplus \Delta \text{MatchSet}^{P_k}) \). Since the emphasis is on incremental calculation, the result of the maintenance routine should preferably be defined as \( \Delta_r = M'_r - M_r \).

The number of parent nodes, the null state memory and the maintenance routine are determined by the node type, along with the way how the subpattern is derived from the subpatterns associated with the parent nodes. Various operations require different node types; they will be discussed later in Section 3.4.

Input nodes are responsible for injecting basic knowledge of the graph model into the Rete net. Their associated subpattern must be \textit{primitive}, i.e. either consist of a single entity constraint of type \( \text{cls} \in \text{Cls} \) (plus a single variable for that entity) or a single relation constraint of type \( \text{fea} \in \text{Fea} \) (plus three \textit{different} variables for relation, source and target). In essence, matching these patterns is equivalent to respectively evaluating the elementary model query operation \( \text{Query}(\ast : \text{cls}) \) or \( \text{Query}(\ast \rightarrow \ast : \text{fea}) \) (see Section 2.1.5). The only significant difference between the match set of the primitive pattern and the corresponding elementary entity (resp. relation) query result is the following: the primitive pattern has a variable associated to the returned entity (resp. returned relation and source / target entity).

Due to the connection between primitive patterns and elementary model queries, the content of input nodes can be determined from the underlying model in a trivial and atomic query step. Thus there is no need for parent nodes, a null state or a node maintenance routine.

Note that the above definition of primitive pattern does not allow the source and target variables of the single relation constraint to coincide, as Section 2.1.5 did not introduce a corresponding elementary model query for self-loop relations. Unlike some graph model formalizations such as [Ren04a], where entity types are represented by self-loop relations, this kind of elementary model query is not vital to graph models formulated as in Section 2.1. Nevertheless, the Rete-based pattern matcher is capable of matching patterns containing self-loops, see Section 3.4.1.3.

\textbf{Definition 49 (Rete input node)} A Rete input node \( r = \langle P_r, Q_r \rangle \) is associated with:

- a primitive subpattern \( P_r \subseteq P \) where \( P \) is a pattern cached by the pattern matcher
- an elementary model query operation \( Q_r \) that is equivalent to the primitive subpattern.

For the sake of uniformity, we can consider a trivial parent set \( \text{Parents}_r = \emptyset \) for any input node \( r \).

\textbf{Definition 50 (Rete net)} A Rete net \( R = \langle N, N^I, M \rangle \) contains a set of Rete nodes \( N = \{r_1, r_2, \ldots, r_n\} \), a subset of which are input nodes \( N^I \subseteq N \) (the rest are operation nodes), and a memory \( M = \{M_{r_1}, M_{r_2}, \ldots, M_{r_n}\} \) composed of the internal memories \( M_r \) of all nodes, each of which is required to contain \( M_r \cdot \text{matches} \), the match set of the associated subpattern. The Rete net is required to be closed and acyclic w.r.t. node-parent relationships.

Note that since node-parent relationships in the Rete net are required to be acyclic, and also closed (\( \forall r \in N : \text{Parents}_r \subseteq N \)), there exists a topological ordering of the nodes so that each node comes after all of its parent nodes.
Definition 51 (Rete-based incremental pattern matcher) The Rete-based pattern matcher $PM_R = \langle S_{Rete}^0, s_{Rete}^0, Match_{Rete}, Maintain_{Rete} \rangle$ based on Rete net $R = (N, N^I, M)$ is a fully cached and incremental stateful pattern matcher that uses the memory of the Rete net as its internal state $s = M$. The Rete-based incremental pattern matcher is consistent at state $\langle G, s \rangle$ iff $M_r.matches = MatchSet_G^{\delta}$ for all nodes $r \in N$.

Since the matcher state is formed of the memories of individual nodes, here $S_{Rete}^0$ is simply the Cartesian product of the possible range of memory contents of the individual nodes, which is the powerset of potential matches of the pattern associated with the node over the given universe $U$.

According to the definitions in Section 3.1.2, a pattern $P$ is (fully) cached iff there exists a node $r(P)$ such that its associated subpattern $P_{r(P)} = P$ equals to the complete pattern. In case of an already cached $P$, it is trivial to retrieve $\langle G, s \rangle.Match(P).out = M_{r(P)}.matches$. On-demand fully caching can be achieved if, in case $P$ is not fully cached yet, $Match(P)$ extends the Rete network (and thus modifies the internal matcher state$^3$) with new nodes so that a production node $r(P)$ and its ancestors are created before yielding the output. Constructing Rete nets for matching a pattern will be discussed in Section 3.4.

Definition 52 (Rete maintenance routine) The matcher maintenance routine $Maintain_{Rete}$ is defined here through its application $\langle G, s \rangle.Maintain_{Rete}(\delta)$. The new matcher state $s' = M' = \langle M'_1, M'_2, \ldots, M'_n \rangle$ is computed using temporary storage $\langle \Delta_1, \Delta_2, \ldots, \Delta_n \rangle$ with semantics $\Delta_e = M'_e.matches - M_e.matches$. The following computations are performed individually for each node, in any order (or even in parallel) that observes the precedence of nodes induced by the node-parent relationship, i.e. the new memories of parent nodes must be available for the maintenance of child nodes. Such a topological ordering exists, since the node-parent relationship is required to be acyclic.

- The maintenance computation for input node $r \in N^I$ consists of first determining $\Delta_e$ by simply filtering $\delta$ to elements relevant for the equivalent elementary query $Q_e$, and then incrementally updating the node memory to the new $M'_e = M'_e.matches := M_e.matches + \Delta_e = \Delta_1 \oplus \Delta_2 \oplus \ldots \oplus \Delta_n$.

- The computation for operation node $r \in N \setminus N^I$, with parents $Parents_r = \langle r_1, r_2, \ldots, r_k \rangle$ already maintained, consists of incrementally calculating $M'_e = M_e.Maintain(M_1.matches \oplus \Delta_1, M_2.matches \oplus \Delta_2, \ldots, M_k.matches \oplus \Delta_k)$ and storing the delta $\Delta_e := M'_e - M_e$.

The consistency of the matcher must be an invariant of $Maintain_{Rete}$, i.e. $\forall r \in N : M_r.matches = MatchSet_G^{\delta}$ implies $\forall r \in N : M_r.matches = MatchSet_G^{\delta}$ or equivalently $\forall r \in N : \Delta_r.matches = \Delta MatchSet_G^{\delta}$. This trivially holds for input nodes, but imposes proof obligations on $Maintain_e$ of each operation node $r$ (with the assumption that all parent nodes already comply).

Finally, the initialization of the Rete net, i.e. determining $S_{Rete}^0$, deserves some technical discussion. The method of initialization described below is not very interesting from the point of view of the thesis, but is included here for the sake of completeness. Each operation node $r$ provided a null state $M^0_r$ which assumed all parent nodes had empty contents; while this is typically true, there can be exceptions (such as the unit node of Definition 55), so the null state has to be adjusted according to the actual, not necessarily empty content of parent nodes (in their adjusted null state), as if that

---

$^3$In this case, the mutable structure of the Rete net must also be part of the pattern matcher state; this was omitted from the previous formalization for brevity
content had just been inserted into the parent nodes. Therefore the matcher null state is defined as follows.

**Definition 53 (Rete null state)** The matcher null state $s^\text{Rete}_0 = (A^0_t, A^0_{t_1}, \ldots, A^0_{t_n})$ is composed of the adjusted null states of each node, where

- the adjusted null state $A^0_t$ of an input node $t \in N^I$ is the empty set; while

- the adjusted null state $A^0_t$ of an operation node $t \in N \setminus N^I$ is derived from the (unadjusted) null state $M^0_t$ as $A^0_t = M^0_t \oplus \text{Maintain}(\emptyset \oplus (A^0_{t_i}.\text{matches} - \emptyset), \emptyset \oplus (A^0_{t_j}.\text{matches} - \emptyset), \ldots, \emptyset \oplus (A^0_{t_n}.\text{matches} - \emptyset))$

This computation, analogously to the net maintenance, can be done in any order that observes the precedence of nodes induced by the node-parent relationship.

### 3.3.4 Discussion of algorithmic complexity

As before, let $\text{Patterns}^\text{Required}$ denote the set of patterns that the Rete-based incremental pattern matcher $\text{PM}_R$ is required to cache. Let $\text{Patterns}^\text{Internal}$ denote any patterns cached at Rete nodes of $\text{PM}_R$ that are not in $\text{Patterns}^\text{Required}$ and are not primitive patterns.

**Query time complexity** for pattern $P \in \text{Patterns}^\text{Required}$. Since $P$ is fully cached, enumerating through all matches takes at most $O(|\text{MatchSet}_P^r|)$ time. Recall that $\Omega(|\text{MatchSet}_P^I|)$ is the lower bound for any pattern matcher.

**Memory complexity** for model $G$. For node $t$, let $\text{OutputSize}_t = |M_t.\text{matches}|$ be the number of partial matches cached at the node, and let $\text{InputSize}_t = \sum_{t' \in \text{Parents}_t} \text{OutputSize}_{t'}$ be the combined size of the partial match sets cached at parent nodes. The memory of a node can cache additional data in addition to the partial matches. However, if $|M_t| = O(\text{OutputSize}_t + \text{InputSize}_t)$, then $t$ is called *locally compact*. If the matcher is consistent and all nodes are locally compact, then the entire Rete net will have a memory complexity of at most $O(\sum_t (\text{OutputSize}_t + \text{InputSize}_t))$; due to consistency this is $O(\sum_t |\text{MatchSet}_G^P|)$. Taking account that cached patterns are either primitive, required or internal, we get $O(|G| + \sum_{P \in \text{Patterns}^\text{Required}} |\text{MatchSet}^P_G| + \sum_{P \in \text{Patterns}^\text{Internal}} |\text{MatchSet}^I_P|)$. Recall that the lower bound for any fully incremental matcher is $\Omega(|G| + \sum_{P \in \text{Patterns}^\text{Required}} |\text{MatchSet}^P_G|)$.

**Maintenance time complexity** for $G \oplus \delta$. For node $t$, let $\text{OutputDelta}_t = |M_t.\text{matches} - M_t.\text{matches}|$ be the size of the match set delta at the node, and let $\text{InputDelta}_t = \sum_{t' \in \text{Parents}_t} \text{OutputDelta}_{t'}$ be the combined size of the match set deltas at parent nodes. Processing input deltas and maintaining the entire memory of a node may not be proportional to the match set delta at the node. However, if $\text{Maintain}_t$ has time complexity of at most $O(\text{OutputDelta}_t + \text{InputDelta}_t)$, then $t$ is called *locally swift*. If the matcher is consistent and all nodes are locally swift, then the entire Rete net will have a maintenance time complexity of at most $O(\sum_t (\text{OutputDelta}_t + \text{InputDelta}_t))$; due to consistency this is $O(\sum_t |\Delta\text{MatchSet}^{P\oplus\delta}_G|)$. Taking account that cached patterns are either primitive, required or internal, we get $O(|\delta| + \sum_{P \in \text{Patterns}^\text{Required}} |\Delta\text{MatchSet}^{P\oplus\delta}_G| + \sum_{P \in \text{Patterns}^\text{Internal}} |\Delta\text{MatchSet}^{I\oplus\delta}_P|)$. This makes the pattern matcher swift. Furthermore, recall that the lower bound for any fully incremental matcher is $\Omega(|\delta| + \sum_{P \in \text{Patterns}^\text{Required}} |\Delta\text{MatchSet}^{P\oplus\delta}_G|)$.

**Discussion.** All Rete nodes that will be discussed in Section 3.4 are locally swift and compact. Therefore Rete only exceeds theoretical lower bounds by the match set size resp. match set deltas of internally cached patterns. These quantities are dependent heavily on the choice of $\text{Patterns}^\text{Internal}$.
which is determined by the quality of the query plan, i.e. the way the Rete net is constructed to cache
the $\text{Patterns}^{\text{Required}}$ patterns. This component of performance is therefore hard to give a general
prediction for.

As Rete is a fully cached matcher, the query times are in stark contrast with non-caching ap-
proaches, where query complexity is often a higher-order polynomial of model size, even for small
match sets. The memory complexity of Rete is higher than many fully cached approaches, especially
minimal cache matchers like TREAT. This is, however, compensated with a maintenance time that is
only proportional to the amount of change, which is predictable based on $\text{Patterns}^{\text{Internal}}$. TREAT,
for instance, does not have this swiftness property, and its maintenance complexity may be a poly-
nomial of the model size, even if the changes have little impact.

3.4 Adapting Rete for graph pattern matching

The following sections will present the adaptation of the overall Rete concept to the specific graph
pattern formalism used in the thesis. Throughout the discussion, concepts of relational al-
gebra [GMUW08] will be used frequently; familiarity with the topic is assumed.

3.4.1 Basic graph pattern matching with Rete

3.4.1.1 Join nodes

To match graph patterns of the core formalism using Rete, it is sufficient to introduce the most basic
kind of beta node type. The join node is a beta node (i.e. it has two parent nodes, the left parent and
the right parent) that computes the relational operation natural join (traditionally denoted by $\Join$) of
the match sets represented by its two parent nodes.

The join node corresponds to a combined subpattern that contains the union of variables of the
subpatterns of parent nodes, and the union of constraints enforced by the parent nodes. The result of
the relational join is the match set of this combined subpattern since the matches in the result will be
exactly those substitutions that correspond to a match of each of the parent nodes (when projected
to their variables).

The following formal definition uses a natural extension of relational join to an operation among
match set deltas: for $\delta^1 = \langle \delta^1_+ , \delta^1_- \rangle$ and $\delta^2 = \langle \delta^2_+ , \delta^2_- \rangle$, their relational join is defined to be $\delta^1 \Join \delta^2 = \langle \delta^1_+ \Join \delta^2_+ , \delta^1_- \Join \delta^2_- \rangle$; where
entire match sets participate in such joins as deltas $\delta = M - \emptyset = \langle M, \emptyset \rangle$. This way, the usual
algebraic properties such as bilinearity will hold, allowing us to simplify the maintenance routine to
an incremental calculation.

Definition 54 (Join node) For left and right parents $\tau_{left}$ and $\tau_{right}$, the join node is a type of oper-
ational node $\tau_{\Join} = \langle P_{\Join}, P_{\text{Parents}}^{\Join}, M_0^{\Join}, M_{\Join}^{\text{matches}} \rangle$, where

- **Parents**: the set of parent nodes consist of the given left and right parent, $P_{\text{Parents}}^{\Join} = \{ \tau_{left}, \tau_{right} \}$, with $P_{\text{left}} = P_{\tau_{left}}$ and $P_{\text{right}} = P_{\tau_{right}}$

- **Pattern**: $P_{\Join} = \langle V, C \rangle$ with $V = V(P_{\text{left}}) \cup V(P_{\text{right}})$ and $C = C(P_{\text{left}}) \cup C(P_{\text{right}})$ (these
do not have to be disjoint unions),

- **Memory**: the memory stores only the match set $M_{\Join} = M_{\Join}^{\text{matches}}$, 

**Invariant:** the consistency invariant is $M_{\alpha_0} = \text{MatchSet}_{\alpha_0} = \text{MatchSet}_{\left.\alpha\right.} \supseteq M_{\alpha_0}$.

**Initialization:** $M_{\alpha_0}^0 = \emptyset \supseteq \emptyset = \emptyset$

**Maintenance:** $M_{\alpha_0}.\text{Maintain}_{\alpha_0}(M_{\left.\alpha\right.}.\text{matches} \oplus \Delta_{\left.\alpha\right.}) = \text{MatchSet}_{\left.\alpha\right.}$ is defined in the following, using notation $M_{\left.\alpha\right.} = M_{\left.\alpha\right.}.\text{matches}$.

It follows easily from the definitions of $M_{\alpha_0}$ and $\text{Maintain}_{\alpha_0}$ that the consistency invariant is preserved, the node is locally compact, and that the maintenance computation is incremental and locally swift.

### 3.4.1.2 Rete net construction

Using the join node type, it is trivial to build a Rete net that matches any simple graph pattern (conforming to the first definition of the concept in Section 2.2.1) that consists only of entity and relation constraints, where the relation constraints are not self-loops. Assume that the pattern to be cached is $P = \langle V, C \rangle$ where $C = C_{\text{ent}} \cup C_{\text{rel}}$ is the set of constraints. For each entity constraint $c = \langle v, t \rangle \in C_{\text{ent}}$ the Rete net contains an input node $r_v = \langle P_v, Q_v \rangle$ with primitive pattern $P_v = \langle V_v, C_v \rangle$ consisting of just this constraint ($V_v = \{v\}, C_v = \{c\}$) and associated elementary query equivalent $Q_v = \text{Query}(\ast : t)$. Similarly for each relation constraint $c = \langle a, v, b, t \rangle \in C_{\text{rel}}$ the Rete net contains an input node $r_c = \langle P_c, Q_c \rangle$ with primitive pattern $P_c = \langle V_c, C_c \rangle$ consisting of just this constraint ($V_c = \{a, v, b, t\}, C_c = \{c\}$) and associated elementary query equivalent $Q_c = \text{Query}(\ast \in c \at \ast)$. Finally, there are $|C| - 1$ join nodes forming a tree structure (w.r.t. node-parent relationships) with the above input nodes as leaves and a single tree root $r$. There are many ways to build the tree, all of which suffice here, although they may differ greatly from a performance point of view (see Section 3.4.4). Now, due to the properties of join nodes, $P_i$ is composed of all constraints and variables of the primitive patterns at the input nodes, therefore $P_i = P$ and $r = r(P)$, i.e. $P$ is fully cached.

**Example 20** The following example elaborates Example 18 on page 47 and Example 19 in more detail. The example relies on (the redefined version of) the *inhibits* pattern, which expresses that a marked place inhibits a transition via an inhibitor arc, as depicted by Figure 3.2 on page 47. The pattern $P_{\text{inhibits}} = \langle V, C \rangle$ is defined on variables $V = \{P, M, K, H, T\}$ with constraints $C_{\text{ent}} = \emptyset$ and $C_{\text{rel}} = \{c_1 = \langle P, M, K, \text{Marking} \rangle, c_2 = \langle P, H, T, \text{Inhibitor-Arc} \rangle\}$. For the sake of brevity, no entity constraints are specified, as $c_1$ and $c_2$ already imply the required types of entity variables $P, K, T$ according to the metamodel of Example 2 on page 11.

The minimal Rete net that matches this pattern is $R = \langle N, N_I, M \rangle$, as depicted before in Figure 3.3 on page 48. Here $N_I = \{r_1, r_2\}$ with input node $r_1$ corresponding to constraint $c_1$, i.e. the memory of $r_1$ contains all marking relations as tuples (matches of $P_{r_1} = \langle \{P, M, K\}, \{c_1\}\rangle$), while $r_2$ contains all inhibitor arcs as tuples (matches of $P_{r_2} = \langle \{P, H, T\}, \{c_2\}\rangle$). $N = N_I \cup \{r_3\}$ contains a single operation node that is a join node with parents $r_1$ and $r_2$. This $r_3$ is associated with $P_{r_3} = \langle \{P, M, K, H, T\}, \{c_1, c_2\}\rangle = \langle V, C \rangle = P_{\text{inhibits}}$, thus the pattern is fully cached. Figure 3.3 shows each Rete node as a grey box that displays its associated pattern, interconnected by grey arrows pointing from parent node to child node.
3.4. ADAPTING RETE FOR GRAPH PATTERN MATCHING

Figure 3.5: Rete net memory contents and deltas for the inhibits pattern

If this Rete net is initialized on the instance model of Example 3 on page 13, nodes will contain in their memories the matches of their associated patterns. This is shown in Figure 3.5, with individual matches represented as white boxes inside the node that contains them (ignore the red and green boxes for now). Input node r₁ will contain the matches of P₁₁, one for each Marking relation: ⟨p₁1, m₁1, k₁₁⟩, ⟨p₁1, m₁2, k₁₂⟩, ⟨p₁1, m₁₃, k₁₃⟩, ⟨p₃, m₄, k₄⟩ (using the tuple notation). Input node r₂ will cache the single InhibitorArc relation: ⟨p₂₂, g₁, t₁⟩. Finally, join node r₃ will compute the relational join of these match sets, which happens to be the empty set. This means that the pattern inhibits has no matches.

If the graph model is changed by the manipulative transaction Fire(t₂) (see Example 6 on page 21), the Rete-based pattern matcher will be maintained according to the graph delta. Figure 3.5 shows matches in the negative delta of a node as red boxes, and matches in the positive delta as green boxes. First, the delta match set of input nodes follows directly from the delta of the graph model; in this case, the delta of r₁ will remove one match and add one match, while the match set of r₂ is unchanged. Then, the delta of the join node r₃ is computed: \( \Delta₃ = (M₁ △ △ Δ₂) + (Δ₂ △ Δ₁) + (Δ₁ △ M₂) = \emptyset + \emptyset + \{+⟨p₂₂, m₅, k₅, g₂, t₂⟩\} \). Therefore one new match appears in the cached match set of pattern inhibits.

3.4.1.3 Corner cases

Note that self-loop relation constraints (where the source and target variables coincide) are disallowed in primitive patterns, therefore the above Rete net construction schema does not work in this case. However, any self-loop relation constraint \( c_{loop} = ⟨a, v, a, t⟩ \in C^{rel} \) can be equivalently substituted with the following two constraints: (i) \( c_{rel} = ⟨a, v, b, t⟩ \in C^{rel} \) for an additional variable \( b \) and (ii) the equality constraint \( c_e = ⟨a, b⟩ \in C^{ent} \). Equality constraints will be dealt with in Section 3.4.2.

A minor detail should not be overlooked here: \( P_r \) will not contain all variables if there are loose variables that are not affected by any entity or relation constraint, and therefore do not appear at the input nodes. While the usefulness of such variables is debatable, it is possible to work around the problem: when building the Rete net according to the above schema, one should assume a special constraint \( c_e = ⟨v, *⟩ \in C^{ent} \) for any \( v \in V^{ent} \) in order to assert that \( v \) should be mapped to an entity, and \( c_e = ⟨a, v, b, *⟩ \in C^{rel} \) for any \( v \in V^{rel} \) in order to assert that \( v \) should be mapped to a relation.
Note that the elementary query equivalents at the corresponding input nodes will be unrestricted by entity or relation type.

As a final special case, the construction process cannot be applied if there are no input nodes; it is not possible to build a tree of joins operations and to provide a final node that caches the pattern. Since each variable, including loose variables, would be involved in at least one input node, this situation can only arise if the pattern has no variables (thus no constraints) at all. The pattern without variables and constraints has a single match that is empty (nullary tuple). While such a pattern by itself is not very useful, Rete construction procedures for more advanced language elements (e.g. NAC, see Section 3.4.2) might require a corresponding Rete node in special cases. Thus a special Rete node called unit node is provided; named for the fact that its constant match set acts as the neutral element for natural join. Due to this property, it could have been included in the previously introduced join tree without effect.

**Definition 55 (Unit node)** The unit node is a type of operational Rete node \( r_0 = (P_0, Parents_0, M_0, Maintain_0) \), where

- **Parents**: there are no parents: \( Parents_0 = \emptyset \),
- **Pattern**: \( P_0 = \langle V, C \rangle \) with \( V = \emptyset \) and \( C = \emptyset \),
- **Memory**: the memory stores only the match set \( M_c = M_c.matches \),
- **Invariant**: the consistency invariant is \( M_0 = MatchSetP_0 = \{()\} \)
- **Initialization**: \( M_0^0 = \{()\} \)
- **Maintenance**: \( Maintain_0 \) is a no-op (returns an empty delta).

### 3.4.2 Rete pattern matching with advanced pattern language features

#### 3.4.2.1 Equality and inequality

Equality and inequality constraints cannot be associated with primitive patterns, as there is no elementary model query to enumerate all pairs of model elements that are respectively equal or different. Although it would be theoretically possible to define such queries, one can easily see that their realization would not be practical (and, in case of attributed models, impossible). In short, we say that these constraints are checkable, but not finitely enumerable. However, a remarkable property of both of these constraints is that it can be decided without knowledge of the graph model whether a given substitution satisfies them or not; these will be called filter constraints.

**Definition 56 (Filter constraint)** A constraint \( c \) expressed over variables \( Dom(c) \) is a filter constraint if it can be associated with a Boole-valued deterministic selector function \( sel_c : \prod^{Dom(c)} \to 2 \) on substitutions of variables \( Dom(c) \), such that for any (partial) substitution \( s \) over any graph model \( G \) with \( Dom(s) \supseteq Dom(c) \), \( s \) satisfies \( c \) iff \( sel_c(s) \) is true.

The standard solution in Rete nets for filter constraints is the alpha node. Alpha nodes are operation nodes that have a single parent node, and their role is to filter the match set of the parent according to the selector function. In terms of relational algebra, alpha nodes perform a selection operation (traditionally denoted by \( \sigma \)).

The following formal definition uses a natural extension of relational selection to an operation among match set deltas: for \( \delta = \langle \delta_+, \delta_- \rangle \), its selection is defined to be \( \delta_{sel} = \langle \delta_{sel}^+, \delta_{sel}^- \rangle \) with
3.4. ADAPTING RETE FOR GRAPH PATTERN MATCHING

\[ \delta^{sel} = \sigma_{sel} \delta_+ \] and \[ \delta^{sel} = \sigma_{sel} \delta_- \]. This way, the usual algebraic properties such as linearity will hold, allowing us to simplify the maintenance routine to an incremental calculation.

Definition 57 (Alpha node for filter constraint) For parent node \( r_{\text{parent}} \) and filter constraint \( c \) with selector \( \text{sel} \), the alpha node is a type of operational node \( r_c = \langle P_c, \text{Parents}_c, M^0_c, \text{Maintain}_c \rangle \), where

- **Parents**: there is a single parent \( \text{Parents}_c = \{ r_{\text{parent}} \} \), with \( P_{\text{parent}} = P_{r_{\text{parent}}} \).
- **Pattern**: \( P_c = \langle V, C \rangle \) with \( V = V(P_{\text{parent}}) \) and \( C = C(P_{\text{parent}}) \cup \{ c \} \).
- **Memory**: the memory stores only the match set \( M_c = M_c.\text{matches} \).
- **Invariant**: the consistency invariant is \( M_c = \text{MatchSet}^{P_c} = \sigma_{\text{sel}} \text{MatchSet}^{P_{\text{parent}}} \).
- **Initialization**: \( M^0_c = \sigma_{\text{sel}} \emptyset = \emptyset \).
- **Maintenance**: \( M_c.\text{Maintain}_c(M_{\text{parent}}.\text{matches} \oplus \Delta_{\text{parent}}) \) is defined in the following, using notation \( M_{\text{parent}} = M_{r_{\text{parent}}}.\text{matches} \), \( \Delta_{\text{parent}} = M_{r_{\text{parent}}}.\text{matches} \), by specifying the delta it calculates:

\[
\Delta_c = M'_c - M_c = \sigma_{\text{sel}} (M_{\text{parent}} + \Delta_{\text{parent}}) - \sigma_{\text{sel}} M_{\text{parent}} = \sigma_{\text{sel}} \Delta_{\text{parent}}.
\]

It follows easily from the definitions of \( M^0_c \) and \( \text{Maintain}_c \) that the consistency invariant is preserved, the node is locally compact, and that the maintenance computation is incremental and locally swift.

The alpha node type enables us to construct Rete nets that match a graph pattern that contains filter constraints in addition to the already discussed entity constraints and non-loop relation constraints. The method of construction is similar to the tree schema described in Section 3.4.1; but for each filter constraint \( c \), an alpha node \( r_c \) has to be inserted (interjected) in the tree between a parent node \( r_{\text{parent}} \) and a child node \( r_{\text{child}} \), such that the variables of \( r_{\text{parent}} \) are sufficient to evaluate the constraint \( V(P_{\text{parent}}) \supseteq \text{Dom}(c) \). For improving performance (e.g. to prefilter both sides of a join), the constructed tree may contain more than one such alpha node for a given filter constraint.

As equality and inequality constraints are examples of filter constraints, we have shown a way of matching them. A complex example involving an inequality filter node will be demonstrated by Example 23 on page 62. Note that no claim is made that an implementation should always follow this approach in constructing a Rete net for patterns with equality constraints. For example, it might be more efficient (due to more specific joins) to consider sets of variables that are connected by equality constraints as a single “unified” variable, and use equality checkers only when necessitated by the elimination (see Section 3.4.1.3) of self-loops.

Since self-loop relation constraints were already shown to be substitutable by a non-loop relation constraint and an equality constraint, the method described so far is sufficient for matching any graph pattern according to Definition 21, with entity constraints, relation constraints (possibly self-loops), equalities and inequalities.

3.4.2.2 Composition

Handling pattern composition \( c^{\text{call}} = \langle P_{\text{called}}, \text{composer} \rangle \) is straightforward, assuming that the called pattern \( P_{\text{called}} \) is already fully cached. As there is a \( r(P_{\text{called}}) \) containing in its memory the matches of the called pattern, it can be included alongside the input nodes as a leaf node. The tree structure of joins and alpha nodes will be built according to the procedure described before. In the terminology introduced before, this means that pattern call constraints are enumerable.
A slight technical hurdle is that in order to be able to participate in the join tree, \(v(P_{\text{called}})\) must be “reinterpreted” / “relabeled” as having the subpattern \(P_{\text{called}}\) with the single constraint \(C(P_{\text{called}}) = \{c^{\text{call}}\}\) and variables \(V(P_{\text{called}}) = \text{Dom}(c^{\text{call}})\). As reinterpretation is a trivial technicality, formal description is omitted here. However, one way to formally describe it is by attaching a reinterpretation node as a child of \(v(P_{\text{called}})\) (such that it copies the contents of its parent node verbatim, but it is associated with the new subpattern), and using this reinterpretation node instead of \(v(P_{\text{called}})\) as the leaf during the tree construction.

### 3.4.2.3 Negative pattern calls

Negative pattern composition asserts that a negative application condition pattern \(\hat{N}_i\) does not have any matches aligned with matches of pattern \(P\). In Section 2.2.2 it was formulated as a separate construct of the formalism (pattern with NAC \(N((P, N^*)\)), and alternatively as a constraint type \(c^{\text{nac}} = \langle \hat{N}_i, \text{composer} \rangle\) analogous to composition. A key observation is that the match set of the pattern with NAC \(\hat{N}_i\) can be expressed using the relational operation anti-join (traditionally denoted as \(\bowtie\)): \(\text{MatchSet}^N = \text{MatchSet}^P \bowtie \text{MatchSet}^{\hat{N}_i}\). In order to be able to construct Rete nets for patterns with NACs, the anti-join node is introduced as a new kind of beta node. For the sake of uniformity with other Rete nodes, here we will use the constraint form of NACs.

Unlike the natural join, relational anti-join is only linear in the first operand: for match sets \(M_{\text{left}}\) and \(M_{\text{right}}\), \((M_{\text{left}} + \Delta_{\text{right}}) \bowtie M_{\text{right}} = (M_{\text{left}} \bowtie M_{\text{right}}) + (\Delta_{\text{right}} \bowtie M_{\text{right}})\). Consequently, it can be extended to an operation where the left operand is a match set delta, but the right operand must remain a match set. However, relational anti-join can be expressed as \(M_{\text{left}} \bowtie M_{\text{right}} = M_{\text{left}} - (M_{\text{left}} \bowtie \pi M_{\text{right}})\) using the relational projection operation \(\pi\) to the set of common variables of \(M_{\text{left}}\) and \(M_{\text{right}}\). Therefore \(M_{\text{left}} \bowtie (M_{\text{right}} + \Delta_{\text{right}}) - M_{\text{left}} \bowtie M_{\text{right}} = (M_{\text{left}} - (M_{\text{left}} \bowtie \pi M_{\text{right}})) - (M_{\text{left}} - (M_{\text{left}} \bowtie \pi M_{\text{right}})) = (M_{\text{left}} \bowtie \pi M_{\text{right}}) - (M_{\text{left}} \bowtie \pi M_{\text{right}} + \Delta_{\text{right}}) = M_{\text{left}} \bowtie (\pi M_{\text{right}} - \pi (M_{\text{right}} + \Delta_{\text{right}})).\) The difference of projections \(\pi M_{\text{right}} - \pi (M_{\text{right}} + \Delta_{\text{right}}),\) in essence a projection of the delta, will be denoted as \(\pi (M_{\text{right}} \oplus \Delta_{\text{right}}),\) and can be calculated incrementally using the appropriate data structures.

#### Definition 58 (Anti-join node)

For negative pattern composition constraint \(c^{\text{nac}} = \langle \hat{N}_i, \text{composer} \rangle\) and left parent \(v_{\text{left}},\) the anti-join node is a type of operational node \(v_b = \langle P_b, \text{Parents}_{b}, M_{b}^0, \text{Maintain}_{b} \rangle,\) where

- **Parents:** the set of parent nodes consist of the given left parent and a right parent that caches the NAC: \(\text{Parents}_{b} = \{v_{\text{left}}, v_{\text{right}}\},\) with \(v_{\text{right}} = v(\hat{N}_i),\) \(P_{\text{left}} = P_{v_{\text{left}}}\) and \(P_{\text{right}} = P_{v_{\text{right}}} = \hat{N}_i\)

- **Pattern:** \(P_b = (V, C)\) with \(V = V(P_{\text{left}})\) and \(C = C(P_{\text{left}}) \cup \{c^{\text{nac}}\}\) (these do not have to be disjoint unions),

- **Memory:** the memory stores only the match set \(M_b = M_{b}.\text{matches},\)

- **Invariant:** the consistency invariant is \(M_b = \text{MatchSet}^{P_b} = \text{MatchSet}^{P_{v_{\text{left}}} \bowtie \text{MatchSet}^{P_{v_{\text{right}}}}}\)

- **Initialization:** \(M_{b}^0 = \emptyset \bowtie \emptyset = \emptyset\)

- **Maintenance:** \(M_b.\text{Maintain}_{b}(M_{\text{left}}.\text{matches} \oplus \Delta_{\text{left}}, M_{\text{right}}.\text{matches} \oplus \Delta_{\text{right}})\) is defined in the following, using notation \(M_{\text{left}} = M_{\text{left}}.\text{matches}, M_{\text{right}} = M_{\text{right}}.\text{matches}, \Delta_{\text{left}} =\)
3.4. ADAPTING RETE FOR GRAPH PATTERN MATCHING

Figure 3.6: Rete net for composite pattern and NAC

\[ \Delta_{\text{left}} \cdot \text{matches}, \quad \text{and} \quad \Delta_{\text{right}} = \Delta_{\text{right}} \cdot \text{matches}, \]

by specifying the delta it calculates:

\[ \Delta_{\varnothing} = M'_{\varnothing} - M_\varnothing = ((M_{\text{left}} + \Delta_{\text{left}}) \circ (M_{\text{right}} + \Delta_{\text{right}})) - (M_{\text{left}} \circ M_{\text{right}}) = (M_{\text{left}} \circ (M_{\text{right}} + \Delta_{\text{right}}) - (M_{\text{left}} \circ M_{\text{right}}) + (\Delta_{\text{left}} \circ (M_{\text{right}} + \Delta_{\text{right}})) = (M_{\text{left}} \circ \pi(M_{\text{right}} \oplus \Delta_{\text{right}})) + (\Delta_{\text{left}} \circ (M_{\text{right}} + \Delta_{\text{right}})). \]

It follows easily from the definitions of \( M_\varnothing \) and \( \text{Maintain}_\varnothing \) that the consistency invariant is preserved, the node is locally compact, and that the maintenance computation is incremental and locally swift.

The anti-join node type enables us to construct Rete nets that match a graph pattern with NAC; its utilization is similar to that of alpha nodes. For a negative composition constraint \( c_{\text{nac}} = \langle N, \text{composer} \rangle \), a corresponding check has to be inserted in the tree as a child of a node \( r \) with \( V(P_r) \supseteq \text{Dom}(c_{\text{nac}}) \); the check will be performed by an anti-join node with left parent \( r_{\text{left}} \).

Analogously to alpha nodes, it is possible to include multiple copies of this check in the tree.

**Example 21** Recalling Example 11 on page 30, pattern unmarkedDisables calls pattern unmarked, and the latter (defined in Example 8 on page 26) has a NAC of its own, namely pattern marked (see Example 7 on page 25). The corresponding Rete nodes are shown here in Figure 3.6. The input node \( r_4 \) enumerates entities of type \( \text{Place} \), while input node \( r_5 \) immediately matches the pattern marked (unnecessary entity constraints were removed as usual). Their anti-join \( r_6 \) (with \( \text{node}_5 \) as right parent / NAC) matches pattern unmarked. Pattern unmarkedDisables is matched in \( r_8 \) by joining input node \( r_7 \) for feature \( \text{InArc} \) against the already constructed \( r_6 \) representing the called pattern.

### 3.4.2.4 Disjunction

In terms of relational algebra, the match set of a disjunctive pattern \( PD = (V, P^*) \) is the union of relational projections onto the set of common variables \( V \) (traditionally denoted by \( \pi_V \)) of the match.
sets of each pattern body \( P_j \in P^* \); formally, \( \text{MatchSet}^{PD} = \bigcup_{P_j \in P^*} \pi_V \text{MatchSet}^P_j \). Note that this union is not guaranteed to be disjoint, as multiple pattern bodies may justify the same match. In fact, recalling the set theoretical definition of relational projection, it should be noted that even a single pattern body may have multiple matches that, when projected onto \( V \), correspond to the same match of the disjunctive pattern.

The node type \textit{projection node} is introduced to address this task; it will use a \textit{multiset} data structure as its node memory to efficiently keep track how many separate derivations are there for a single match.

\textbf{Definition 59 (Projection node) for disjunctive pattern} \( PD = (V, P^*) \) with bodies \( P^* = \{ P_1, P_2, \ldots, P_k \} \), the projection node is a type of operational node \( r_\pi = (P_\pi, \text{Parents}_\pi, M^0_\pi, \text{Maintain}_\pi) \), where

- \textbf{Parents:} the set of parent nodes consist of the nodes caching the pattern bodies, \( \text{Parents}_\pi = \{ r_1, r_2, \ldots, r_k \} \), with \( P(t_j) = P_j \) for each of them,

- \textbf{Pattern:} \( P_\pi = PD = (V, P^*) \),

- \textbf{Memory:} the memory stores a \textit{multiset} of matches: \( M_\pi = M_\pi.\text{counter} : (U \times U \times \ldots \times U) \rightarrow \mathbb{N} \); this of course contains the match set \( M_\pi.\text{matches} = \text{Dom}(M_\pi.\text{counter}) \) (where the domain of a multiset is defined as the elements mapped to a non-zero value),

- \textbf{Invariant:} the consistency invariant is that occurrences of matches \( m : V \rightarrow U \) are counted by the multiset: \( M_\pi.\text{counter} = m \mapsto \sum_{P_j \in P^*} |\text{ResultSet}^P_j(m)| \); this implies \( M_\pi.\text{matches} = \text{MatchSet}^P = \bigcup_{P_j \in P^*} \pi_V(\text{MatchSet}^P_j) \),

- \textbf{Initialization:} \( M^0_\pi.\text{matches} = \bigcup_{P_j \in P^*} \pi_V\emptyset = \emptyset \), therefore \( M^0_\pi.\text{counter} = \emptyset \) as well,

- \textbf{Maintenance:} \( M_\pi.\text{Maintain}_\pi(M_{t_1}.\text{matches} \oplus \Delta_{t_1}, \ldots, M_{t_k}.\text{matches} \oplus \Delta_{t_k}) \) is defined in the following, using notation \( M_j = M_{\pi_j}.\text{matches} \) and \( \Delta_j = (\delta_j^+, \delta_j^-) = M_{\pi_j}.\text{matches} \); by specifying the delta it calculates: \( \Delta_\pi = M_{\pi_\pi}.\text{counter} - M_{\pi_\pi}.\text{counter} = m \mapsto (\sum_{P_j \in P^*} |\{ m' \mid m' \in (M_j + \Delta_j) \land m \subseteq m' \}| - |\{ m' \mid m' \in M_j \land m \subseteq m' \}|) = m \mapsto (\sum_{P_j \in P^*} |\{ m' \mid m' \in \delta_j^+ \land m \subseteq m' \}| - |\{ m' \mid m' \in \delta_j^- \land m \subseteq m' \}|) \), i.e. the counters of the multiset are incremented by each match in the positive delta and decremented by matches of the negative delta.

It follows easily from the definitions of \( M^0_\pi \) and \( \text{Maintain}_\pi \) that the consistency invariant is preserved, the node is locally compact, and that the maintenance computation is incremental and locally swift.

Now disjunctive patterns can be matched by constructing separate Rete trees (using the approach introduced in earlier sections) for each pattern body, and then finally using a projection node that projects each body to the set of parameter variables to obtain the match set of the disjunctive pattern.

\textbf{Example 22} Recalling Example 15 on page 32, disjunctive pattern \texttt{dисables} has two bodies, patterns \texttt{unmarkedDisables} and \texttt{inhibits}. Example 20 and Example 21 show how Rete nodes are constructed to match each of the bodies. Assuming all these Rete nodes are built in a single net, Figure 3.7 shows how a projection node \( r_0 \) computes the union of the projected match sets of both bodies to obtain the match set of the disjunctive pattern.
3.4. ADAPTING RETE FOR GRAPH PATTERN MATCHING

3.4.3 Rete pattern matching with attributes

3.4.3.1 Data entity, data relation and value literal constraints

Matching attributed patterns in general is impossible for any finite pattern matcher, since data entity and data relation constraints are not enumerable. The previously introduced Rete building mechanism will fail: with no corresponding elementary model query, as it was already established in Section 2.1.5, data entity and data relation constraints are not suitable as primitive patterns for input nodes. The narrower set of matchable graph patterns (see Definition 31), however, will be handled in the following paragraphs.

For a data predicate constraint \( c = \langle a, v, b, t \rangle \in C^{rel}_{Dat} \), a corresponding primitive pattern \( P_c \) could be defined, although it cannot be included as input node in a Rete net, as the corresponding elementary model query cannot be finitely evaluated. However, in case of an enumerable data predicate (see Definition 18), the data predicate constraint can be matched by a compute node instead, taking advantage of the corresponding computable query:

**Definition 60 (Compute node)** For a data predicate constraint \( c = \langle a, v, b, t \rangle \in C^{rel}_{Dat} \) where \( t \) is a data predicate enumerable by source, and parent node \( r_{parent} \) with \( a \in V(P(r_{parent})) \) (i.e. it contains values for the source element), the compute node is a type of operational node \( r_{comp} = \langle P_{comp}, Parents_{eval}, M_{comp}^0, Maintain_{comp} \rangle \), where

- **Parents**: the set of parent nodes consist of the given single parent, \( Parents_{eval} = \{ r_{parent} \} \), with \( P_{parent} = P_{r_{parent}} \).
- **Pattern**: \( P_{comp} = \langle V, C \rangle \) with \( V = V(P_{parent}) \cup \{ v, b, t \} \) and \( C = C(P_{parent}) \cup \{ c \} \).
- **Memory**: the memory stores only the match set \( M_{comp} = M_{comp,.matches} \).
- **Invariant**: the consistency invariant is \( M_{comp} \bowtie MatchSet^{P_{comp}} = MatchSet^{P_{r_{parent}} \bowtie MatchSet^P} \) (even if the latter is not finitely enumerable).
- **Initialization**: \( M_{comp}^0 = \emptyset \bowtie MatchSet^P = \emptyset \).
**Maintenance**: $M_{\text{comp}}.\text{Maintain}_{\text{comp}}(M_{\text{parent}}, \text{matches} \oplus \Delta_{\text{parent}})$ is defined in the following, using notation $M_{\text{parent}} = M_{\text{parent}} . \text{matches}$, $M_c = M_c . \text{matches}$ and $\Delta_{\text{parent}} = \Delta_{\text{parent}} . \text{matches}$, by specifying the delta it calculates: $\Delta_{\text{comp}} = M'_{\text{comp}} - M_{\text{comp}} = (M_{\text{parent}} + \Delta_{\text{parent}}) \bowtie M_c - (M_{\text{parent}} \bowtie M_{\text{right}}) = \Delta_{\text{parent}} \bowtie M_c$.

An analogous node can be constructed if the data predicate is enumerable by target.

The expression $\Delta_{\text{parent}} \bowtie M_c$ is finitely computable as all delta tuples $m$ in $\Delta_{\text{parent}}$ contain a value for variable $a$, and thus it will be joined against $\text{Dat}.\text{Query}(m(a) \xrightarrow{\text{t}} \ast).\text{out}$ which is computable since $t$ is enumerable by source. Furthermore, it follows easily from the definitions of $M_{\text{comp}}^0$ and $\text{Maintain}_{\text{comp}}$ that the consistency invariant is preserved, the node is locally compact, and that the maintenance computation is incremental and locally swift.

Constraints of attributed patterns can be matched in the following way:

- **Structural entity and relation constraints** ($C_{\text{Str}}^{\text{ent}}$ and $C_{\text{Str}}^{\text{rel}}$) correspond to input nodes, as before.

- It is easy to see that value assignment attribute constraints ($C_{\text{str}}^{\text{rel}}$) are also enumerable.

- For value literal constraint $c = (v, x)$, one can construct input nodes associated with the equivalent elementary model query $\text{Query}(x : \ast)$; these input nodes can be used in the construction process analogously to the input nodes for entity and relation constraints.

- An attribute type constraint $c = (v, t) \in C_{\text{Dat}}^{\text{ent}}$ expresses that the substituted value of variable $v$ is of the data type $t$, which can be determined independently of the current graph model (as the immutable data algebra $\text{Dat}$ is known a priori). Therefore $c$ is a filter constraint with selector $sel_c = s \mapsto (\text{Dat} \models s(v) : t)$, and can be handled like any other filter constraint.

- A data predicate constraint $c = (a, v, b, t) \in C_{\text{Dat}}^{\text{rel}}$ can also be considered a simple filter check like attribute type constraints above. Alternatively, if the data predicate is enumerable by source or target, the constraint can be matched by interjecting in the tree a compute node as a child of a node that contains the variable $a$ respectively $b$.

- Further elements of the pattern language can be supported in the same way as described before.

Finally, all that is left to show is that the Rete construction procedure successfully reaches a point where attribute type and data predicate constraints can be evaluated (using a filter node or compute node), since there are Rete nodes that contain the required variables. This follows (inductively) from the definition of matchable patterns (see Definition 31). Each entity variable is assignable. Therefore either there is an associated value literal / value assignment constraint, or else there must be an associated data predicate constraint that is enumerable by some assignable variable(s). In the first case, the variable appears in at least one input node. In the latter case, the variable can be introduced by a compute node whose required variable (established to be assignable) can be made available in some suitable parent node.

**Example 23** Figure 3.8 shows the Rete net built for the left-hand-side pattern of the attributed version of the GT rule `removeToken`, introduced in Example 17. Node $v_1$ is a value assignment input node capturing the attributed version of the `Marking` type. Node $v_2$ is a value literal input node containing 0 as the single value of Z. Node $v_3$ joins this zero value alongside each `Marking` instance. Next, filter node $v_4$ checks whether $X \neq Z$, i.e. the marking value is non-zero; so far this is the attributed version of the pattern marked from Example 10. Finally, knowing that data predicate
Successor is functionally determined by target, compute node \(r_5\) computes the single value \(Y\) for which \(X\) is the next integer. This completes the LHS pattern; each match cached at \(r_5\) is now ready for the application of the GT rule \(\text{removeToken}\) (i.e., redirecting \(MX\) to \(Y\)).

3.4.3.2 Aggregate constraints

Aggregate constraints, like certain kinds of data predicate constraints, are only enumerable by certain variables (see Section 2.2.2.4 on page 31), thus it is subject to restriction associated with matchable patterns (see Definition 31). Constructing a Rete net for a matchable pattern containing an aggregate constraint can be achieved by the same procedure that was presented in Section 3.4.3.1 for attributed patterns, where aggregation is treated similarly to functionally determined data predicates. Instead of a compute node, however, an aggregate node is used:

**Definition 61 (Aggregate node)** For a match aggregate constraint \(\mathcal{C}^{aggregate} = \langle P_{\text{called}}, \text{composer, aggregator, } v_{result} \rangle \in \mathcal{C}^{aggregate}\) with aggregator = \(\langle \text{mapper, reducer, unmapper} \rangle\) containing reducer = \(\langle A, \otimes \rangle\), and left parent \(r_{left}\), the aggregate node is a type of operational node \(r_{aggregate} = \langle P_{aggregate}, Parents_{aggregate}, M_0^{aggregate}, Maintain_{aggregate} \rangle\), where

- **Parents**: the set of parent nodes consists of the given left parent and a right parent that caches the called pattern: \(Parents_{aggregate} = \{ r_{left}, r_{right} \} \), with \(r_{right} = r(P_{\text{called}}), P_{left} = P_{r_{left}}\)
CHAPTER 3. INCREMENTAL GRAPH PATTERN MATCHING

and \( P_{right} = P_{right} = P_{called} \)

- **Pattern:** \( P_{aggregate} = \langle V, C \rangle \) with \( V = V(P_{left}) \cup \{v_{result} \} \) and \( C = C(P_{left}) \cup \{c_{aggregate} \} \) (these do not have to be disjoint unions),

- **Memory:** in addition to the match set, the memory stores a map from matches of the left parent to elements of the reducer group: \( M_{aggregate} = \langle M_{aggregate}.matches, M_{aggregate}.reduced \rangle \) with \( M_{aggregate}.reduced : (U \times U \times \ldots \times U) \rightarrow A \); the match set is derived as \( M_{aggregate}.matches = \{ \{m_{left} \} \bowtie \{\{v_{result} \mapsto \text{unmapper}(a)\}\} | \langle m_{left}, a \rangle \in M_{aggregate}.reduced \land \text{unmapper}(a) \neq \bot \} \),

- **Invariant:** the consistency invariant is defined not on the matches, but on \( M_{aggregate}.reduced \) (from which the matches can be trivially derived) as \( M_{aggregate}.reduced = \{ \{m_{left}, \text{unmapper}(\prod_{m_{right} \in \text{incidents}} \text{mapper}(m_{right}))\} | m_{left} \in \text{MatchSet}^{P_{left}} \} \) where incidents is defined as before,

- **Initialization:** \( M_{aggregate}^0 = \emptyset \) since \( \text{MatchSet}^{P_{left}} = \emptyset \).

The lengthy formalization of \( \text{Maintain}_{aggregate} \) is omitted here for brevity, but it is fairly easy to explain intuitively: new matches of \( P_{left} \) are \( \text{composer} \)-joined against \( \text{MatchSet}^{P_{right}} \), the results are mapped to \( A \) by \( \text{mapper} \), reduced along \( \otimes \) (which yields \( 1_\otimes \) if there are no joining matches) and stored in \( M_{aggregate}.reduced \); then it is mapped back by \( \text{unmapper} \) into the value of \( v_{result} \) in the new match of \( P_{aggregate} \) (or, if \( \bot \), there is no new match). Removed matches of \( P_{left} \) are purged from \( M_{aggregate}.reduced \) and thus from \( M_{aggregate}.matches \) (if it was present). The delta of \( \text{MatchSet}^{P_{right}} \) is mapped by \( \text{mapper} \), then \( \text{composer} \)-joined against \( P_{left} \), then each insertion in this delta will \( \otimes \)-multiply the corresponding entry stored in \( M_{aggregate}.reduced \), and each removal will \( \otimes \)-multiply it with the \( \otimes \)-inverse; afterwards, \( M_{aggregate}.matches \) is updated accordingly.

**Example 24** In the non-attributed version of the metamodel, tokens are individual entities, each attached by a \textit{Marking} relation to a place. Figure 3.9 shows a Rete net that counts the number of tokens at a place, thereby deriving a substitute of the integer-valued \textit{Marking} feature of the attributed version of the metamodel.

The \textit{Marking} relations to be counted are contained in \( r_2 \). At the same time \( r_1 \) simply enumerates \textit{Place} instances. Their child node \( r_3 \) is an aggregate node joining left parent \( r_1 \) against right parent \( r_2 \) by common variable \( P \). The aggregator is \textit{aggregator\_count} (see Example 12 on page 31), and the result is captured as new variable \( X \). There will be one match in \( r_3 \) for every \textit{Place}, consisting of the place itself and an integer \( X \) indicating the number of its tokens.

3.4.4 Realization considerations

3.4.4.1 Performance

When building an actual software implementation of the Rete approach, many low-level practical decisions have to be made. The choices can have a major impact on performance.

Some important aspects will be addressed below. Further issues specific to Rete implementations are discussed in literature, see for example [Doo95].
3.4. ADAPTING RETE FOR GRAPH PATTERN MATCHING

Query plan. As one important degree of freedom, the Rete construction procedures outlined throughout Section 3.4 do not specify the exact join order of the tree, and the placement of alpha nodes, compute nodes, anti-join nodes etc. within the tree; in short, the query plan is subject to optimization. Rete is not a minimal cache algorithm; while storing the match set of fully cached patterns is unavoidable, the size of match sets of subpatterns can vary greatly. As already discussed in Section 3.3.4, the choice of Internal patterns has a large influence on both memory consumption and maintenance overhead. Therefore a reasonable implementation should employ some strategy that aims at keeping these auxiliary match sets as small as possible, thereby reducing memory footprint and maintenance time. Match set sizes are difficult to predict without actually matching the pattern, but various optimization heuristics can be applied, such as avoiding joins where the two parent nodes have no variables in common.

Node sharing. A Rete net can cache multiple patterns, some of which may use the same classifiers or features. Even a single pattern may have multiple entity or relation pattern constraints using the same classifier or feature. Therefore there will be several input nodes whose primitive patterns are labeled with different variable names, but their elementary query is the same. This redundancy can be eliminated if these nodes share their memory, which is maintained in a single coordinated action. An equivalent view of node sharing is to treat these input nodes as a single node, with multiple “facets” characterised by different patterns. This is analogous to the “reinterpretation” / “relabeling” used for pattern composition (see Section 3.4.2). Node sharing can be propagated to child nodes; e.g. if two join nodes have the same pair of parents (parents that are facets of the same node memory) and they join on the same variables (or at least variables consistent with the relabeling between facets), then the join nodes themselves can be considered as multiple facets of the same node with a single memory. There are techniques associated with Rete [BGT91] that build the query plan in a way that it attempts to maximize node sharing among operation nodes.

Indexing. The performance of join operations can be greatly enhanced if the content of a parent node is indexed according to the value(s) of variables(s) that are shared between both parent nodes. Using the notations from Definition 54, instead of calculating $\Delta_{left} \bowtie M_{right}$ by a nested iteration through both sets, it is more efficient to iterate through $\Delta_{left}$ (where a delta is frequently much smaller than a whole match set) and look up corresponding tuples via the index structures built on top of $M_{right}$. Such an index structure can be stored as part of the memory associated with node
\textit{r}_{right}, \textit{e.g.} \textit{M}_{right}.\textit{index}, in addition to \textit{M}_{right}.\textit{matches}. The size of such indices is asymptotically bounded by the size of the match set, therefore memory complexity is not increased.

**Memoryless nodes.** An important observation is that several kinds of operation nodes compute their stored match set (and its deltas) from the match set in their parent incrementally, only using the deltas of the parent node (and index structures, in case of a join). This means that after the initial Rete construction, the match set of the parent node is never read again. Furthermore, the content of input nodes can always be reconstructed using the elementary model queries. Many nodes can therefore be \textit{memoryless}, \textit{i.e.} they do not need to store their match sets in their memory, thus the size of the matcher internal state \textit{s} can be reduced. An important exception is nodes associated with a pattern that is required to be fully cached. Note that this improvement does not make Rete a minimal cache matcher, as many internal nodes have to maintain one or more index structures (see above) for the sake of join/anti-join child nodes.

**Redundant constraints.** Constraints that are enforced by a given node according to the node type, may themselves entail additional constraints according to reasoning based on type inference or metamodel-specific well-formedness rules. For example, an input node for a relation constraint additionally enforces entity constraints: the source entity must instantiate the owner of the feature, while the target must instantiate the range. In these cases, it is safe to add the inferred constraints to the pattern associated with the node, if it is required by the pattern to be matched. This way, building input nodes, alpha nodes, etc. for certain pattern constraints can be avoided, reducing the size of the Rete net. The examples throughout Section 3.4 assumed for the sake of brevity that the pattern does not contain such redundant constraints; but even if it did contain such constraints, it could be matched by the same Rete net structure.

### 3.4.4.2 Integration

A further implementation question is how the Rete-based pattern matcher is integrated with the underlying modeling technology (such as the platforms introduced in Section 2.1.6). The modeling layer needs to provide the following services:

**Elementary model queries** In order to be able to populate the caches, the pattern matcher needs to be able to execute the elementary model queries associated with input nodes. There are no such requirements for the operation nodes, as they derive their contents from their parent nodes. The efficiency of these model queries may vary greatly between modeling platforms. In the ViATRA2 implementation of VPM, enumerating all instances of an entity/relation type is very efficient, while EMF requires a full traversal of the entire model to identify which elements instantiate the given type. Full model traversals are expensive for large models, therefore the integration solution either has to ensure that all input nodes are initialized during a single traversal of the EMF model, or that a suitable index structure is initialized in a single traversal, after which input nodes may be efficiently populated at any time using the index. Note that constructing such a model-wide index has additional memory requirements, and upon changes to the model the index will have to be maintained afterwards. However, one can argue that such a stateful integration is required to extend EMF capabilities so that it can provide the required elementary model query operations of Section 2.1.5; it can be regarded as a useful abstraction layer to hide the idiosyncratic details of the modeling platform.
3.5. Performance Evaluation

Change notifications The other required interface between the model and the matcher is the maintenance phase. The modeling platform must be able to provide notifications of its change deltas whenever the model undergoes any kind of change, so that the maintenance routine of the stateful matcher can be called. Once again, it may be beneficial to use an abstraction layer to translate change notifications specific to the modeling platform into deltas consumed by the Rete-based pattern matcher. For instance, when a new containment subtree is attached to the model in case of EMF, such a translation is required to obtain individual model element insertions from the EMF notification which contains a reference to the root of the subtree only, along with the description of where it is inserted.

3.5 Performance evaluation

In the following, I will present an experimental evaluation that contrasts the performance of incremental pattern matching against non-incremental approaches in a model simulation scenario. In particular, the VIATRA2-based implementation [23] of the Rete algorithm was compared against the original non-incremental matcher [VHV08] of VIATRA2, as well as GrGEN.NET [JBK10], which is considered to be one of the fastest graph transformation engines.

The presented measurement is only one among many experiments [23,16,14,2,11] that have been conducted under different workloads and scenarios (including model-to-model transformation and well-formedness checks) in order to find the conditions under which incremental pattern matching is beneficial.

This section follows [16]. The description of the benchmark is available in Section 3.5.1, followed by the presentation of measurement results in Section 3.5.2. Finally, Section 3.5.3 summarizes the related work.

3.5.1 Petri net firing: a model simulation benchmark

Description. The scenario of simulating visual languages with dynamic operational semantics is a frequent and important use case for graph models and graph pattern matching. Therefore Petri net simulation was selected [16] as a representative performance test from this category.

This scenario involves typical domain specific language simulation with the following characteristics: (i) mostly static graph structure, (ii) relatively small and local model manipulations, and (iii) typical as-long-as-possible (ALAP) execution mode. The workload focuses on the effective reusability of already matched elements as typical firing of a transition only involves a small part of the net. While an incremental pattern matcher can track the changes of the Petri net and update the involved partial matches only, non-incremental local search based approaches will have to restart the matching from scratch after the net changed.

Test case generation. In the Petri net test set, “regular” Petri nets were selected as test cases, which are generated automatically. Here regular means that the number of places and transitions are approximately equal. Furthermore, the net has only a low number of tokens, and thus, there are few fireable transitions in each marking.

The elements of the test set were generated from a small initial net by repeatedly applying six growth operations. These growth operations are defined by reversing the reduction operations that were described in [Mur89] as means to preserve safety and liveness properties of the net. At each iteration of the Petri net generation process, a growth operation was selected by weighted random
CHAPTER 3. INCREMENTAL GRAPH PATTERN MATCHING

Sampling; this allows fine parametrization of the number of transitions and places with an average fan-out of 3-5 incoming and outgoing edges. In all test cases, the generation started from the Petri net depicted in Figure 3.10 (which is trivially a live net) and the final test graphs are available in PNML [JKW02] format at [PNB08]. As the size of a Petri net cannot be described by only a single parameter, generated Petri nets were characterized by the number of growth operations that were applied to produce the model, indicative of the relative ”size” of test cases.

Figure 3.10: The initial Petri net from which the test cases were grown

**Execution.** A step in the iterative execution sequence contains two phases: (i) first a fireable transition $t$ is randomly selected from the match set of pattern fireable (see Example 16) and then (ii) the manipulation operation $\text{Fire}(t)$ (see Example 6) is applied to change of token markings according to Petri net firing semantics.

Despite its simple execution semantics, it is easy to derive additional Petri nets as new benchmark scenarios with significantly different run-time characteristics for the different graph transformation tools. For example, a Petri net with an equal number of transitions, places and tokens but with few fireable transitions can be used as a benchmark where type-based search space reduction strategies of pattern matcher algorithms are neutralized, which forces the pattern matchers to use other heuristics.

Note that the only assumption on these Petri net test cases is to use live and bounded nets to have a potentially unbounded execution sequence. In the benchmark runs, 1 000 consecutive transition firings were designated as Short execution sequences and 1 000 000 transition firings as Long execution sequences.

**Benchmarking.** For this benchmark, the total execution time of the simulation sequences are compared. As the actual selection of transitions to be fired are non-deterministically determined by the tools, different tools (and indeed different test runs with a single tool) may select their own execution paths, but this random factor turned out to have only insignificant effects on execution times.

More benchmarks for incremental pattern matching are presented in [16].

### 3.5.2 Measurement results

The measurements reported here have been carried out in 2008, on a standard desktop computer with a 2 GHz Intel Core2 processor with 2 gigabytes of system RAM available, running version 1.6.0_05 of the 32-bit Sun Java SE Runtime (for VIATRA2) and version 3.0 of the .NET Framework on Windows Vista (for GrGEN.NET). Ten test runs were executed, and the results were calculated by averaging the values excluding the highest and lowest number. The transformation sequences were implemented so that little or no textual output was generated (lest its emission should dominate execution time); but in the case of VIATRA2, the GUI was not completely disabled. Execution times were measured with millisecond precision as allowed by the operating system calls.
The Petri net simulation benchmark was executed with short (1,000 fired transitions) and long (1,000,000 firings) execution sequences.

<table>
<thead>
<tr>
<th>Net Size</th>
<th>Places</th>
<th>Transitions</th>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>7497</td>
<td>7450</td>
<td>10</td>
</tr>
<tr>
<td>20000</td>
<td>14987</td>
<td>14870</td>
<td>10</td>
</tr>
<tr>
<td>50000</td>
<td>37581</td>
<td>37593</td>
<td>10</td>
</tr>
<tr>
<td>75000</td>
<td>56331</td>
<td>56053</td>
<td>10</td>
</tr>
<tr>
<td>100000</td>
<td>74924</td>
<td>75124</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 3.11: # growth operations applied ("net size") vs. # model elements in test cases

As it can be seen from the graph, VIATRA/RETE scales predictably up to model size of $10^5$ with a speed of at least two orders of magnitude faster than VIATRA/LS.

This difference is the consequence of the advantage that the incremental approach does not need to re-evaluate patterns on the whole model after each manipulative transaction (i.e. transition firing), only on the delta. In fact, the execution time is seemingly almost a constant function of model size, due to the property that the average size of the local context of the model delta of a firing (hence the cost of the maintenance routine) does not grow significantly with model size.

As expected, the incremental approach works well for large model sizes as long as there is enough memory (the spike in case of long transformation sequences occurred because of garbage collection as the heap was exceeded).

VIATRA/RETE matches and outperforms the GrGEN.NET tool for very large models in case of both short and long execution sequences. This result is a significant achievement considering
the architectural and run-time differences between VIATRA2 and GrGEN.NET. Most notably, GrGEN.NET uses compile-time optimizations and an entirely different model persistence approach based on compile-time generated type information, whereas VIATRA2 uses a generic model storage supporting dynamic typing and support for interactive applications such as a notification and transaction management mechanism (note that the VIATRA2 GUI was not disabled for the measurement, while GrGEN.NET was used without GUI through GrShell). However, for fairness, it should be pointed out that the GrGEN.NET implementation of this benchmark was prepared by the author (i.e. by a GrGEN non-expert), thus additional language or tool-specific optimizations might be available. No such manual fine-tuning was applied for VIATRA2 either.

The results show that incremental matching performance is superior in case of the specific workload induced by the Petri net firing case study. Overall, the results clearly demonstrate the viability of the incremental approach in analogous model simulation scenarios: very fast execution with predictable, linear scaling up to memory limitations.

3.5.3 Related work in graph transformation benchmarking

The presented benchmark is an excerpt from [16], which aims to design and evaluate graph transformation benchmark cases for the purpose of measuring the performance of incremental approaches. These scenarios are conceptual continuations of the comprehensive graph transformation benchmark library proposed earlier in [VSV05a] (described more extensively in [VSV05b]), which gave an overview on typical application scenarios of graph transformation together with their characteristic features.

[GK07] suggested some improvements to the benchmarks described in [VSV05a] and reported measurement results for many graph transformation tools including AGG [ERT99], PROGRES [Sch90], Fujaba [NNZ00], and GrGEN.NET [JBK10].

A similar approach to graph transformation benchmarking was used for the AGTIVE Tool Contest [The07], including a simulation problem for the Ludo table game. The Petri net firing test case that was presented here is better suited for benchmarking performance since it can be parameterized to scale up to large model sizes and long transformation sequences. Later, the GraBaTs tool contest [Gra08] introduced an AntWorld case study [Alb08] that had similar properties; see [2] for an experimental assessment of the Rete implementation on this benchmark.

3.5.4 Performance discussion

For more investigation of cases where incremental matching is beneficial or disadvantageous, see papers [23,14,2]. The general observation is that as long as there is sufficient available memory to fit the non-minimal incremental caches of Rete, choosing the incremental strategy usually pays off in execution time.

There are some exceptions, when incremental approaches are not beneficial. If there is very intensive model manipulation (such as the replacement of a significant portion of the model) between infrequent queries, the update overhead may eventually offset the benefits of instantaneous queries. Additionally if queries are not invoked regularly (“fire-and-forget”), then constructing the caches was pointless as well. This is especially true if the query was only invoked in seeded pattern matching, and there was no need for the entire match set to be computed in the first place.

Memory overhead is proportional to the size of the match sets cached at Rete nodes. This means that the memory cost of Rete may be prohibitively high at large models if patterns are defined in a way that their match sets grow superlinearly in model size. Rete is not a minimal cache matcher,
thus match sets of subpatterns cached at nodes may potentially dominate memory footprint, even if user-specified patterns are expected to have small match sets.

On the other hand, the various measurements have identified frequently occurring scenarios where incremental matching has great benefits due to frequent model queries with limited change inbetween. They include the use case of on-the-fly model validation (checking of design rules, well-formedness constraints) with instantaneous feedback; the simulation or state space exploration of behavioral models (such as the Petri net example above); and model-to-model transformation, especially in a live and incremental synchronization scenario.

In some cases, the best performance can be achieved by combining pattern matching strategies [14,2].

3.6 Chapter conclusions

The current chapter has formalized incremental pattern matching, adapted the Rete algorithm for incrementally matching the rich pattern language used in the thesis, and demonstrated the performance benefits of the approach. This is the central contribution that will serve as the backbone of all other scientific results presented in the next chapters. Chapter 4 will introduce three separate novel extensions to this core technique. Chapter 5 will show how this approach for model queries can be realized in context of an industrial technology platform. Chapter 6 will apply these results in incremental pattern matching to the domain of security requirement engineering. Finally, Chapter 7 will generalize the concept of graph patterns to change-driven model queries, and provide evaluation strategies built upon the Rete-based technique discussed here.
Chapter 4

Advanced Incremental Pattern Matching

This chapter presents three separate novel extensions to the core incremental pattern matching approach introduced in Chapter 3. First, Section 4.1 shows ways to improve execution performance by taking advantage of the parallel processing capabilities of modern hardware. As an important extension to the non-recursive graph pattern language that was addressed so far, Section 4.2 introduces transitive closure as a query language element, and presents a demonstratedly efficient evaluation technique. Finally, Section 4.3 discusses experiments on incremental matching within relational databases.

4.1 Incremental pattern matching on multi-core platforms

This section partially follows [4] and [36].

4.1.1 Introduction

Nowadays, a main challenge of software engineering is the adaptation to parallel computing architectures. In order to increase execution speed, algorithm designers need to think of new ways to exploit the computing power of multi core processors instead of purely relying on more efficient processor designs. Experience has shown that this is a complicated task: whether parallel execution can actually be effectively applied depends largely on the problem itself.

Model transformation is an application domain where speed optimization based on parallel execution has a lot of potential, especially in case of large, industrial models. In fact, model transformations seem to be an ideal target for parallel execution as in practical transformations, many similar, or almost identical model structures need to be traversed and transformed. Frequently, these model manipulation sequences are non-conflicting, which naturally calls for an execution model where these sequences are executed on the available processors in parallel.

However, parallelization strategies must be revised if incremental pattern matching (see Chapter 3) is applied, since the latter offers an entirely different execution model compared to local search-based pattern matcher approaches traditionally found in literature. The LHS matching phase of GT execution is reduced to fast read-from-cache operations, in exchange for the overhead imposed by maintenance phases which are triggered by the model manipulation operations on parallel transfor-
mation threads. Therefore in case of parallel GT execution, incremental pattern matching raises the following additional issues:

1. pattern matcher speedup is only expected from the parallelization of the maintenance routine, as opposed to the cheap match queries;

2. the implementation of the stateful pattern matcher must cope with model manipulation and model queries on separate threads of execution, if model manipulation is made parallel;

3. interaction with the stateful pattern matcher may influence (i.e. delay) the execution of model manipulation thread(s), possibly diminishing the expected performance benefits.

I propose a novel solution that aims to address these issues. First, Section 4.1.3 will discuss in detail how cache maintenance phases may be executed concurrently to the main execution thread of the model transformation. In this case, model manipulation (or textual output emission, e.g. in the case of code generation transformations) may continue uninterrupted while the Rete memories are maintained concurrently in a separate thread. This approach aims to reduce the execution time overhead imposed by the matcher maintenance routine, in the case when parallel computing power is available.

Then, if further scaling up is required, Section 4.1.4 generalizes this approach by a multi-threaded maintenance routine that can take advantage of multiple processing cores by itself. It is important to point out that both of these approaches are significantly different from parallel pattern search.

Finally, the proposed pattern matcher can also be applied to a multi-threaded model manipulation context to let the model manipulation transactions take advantage of the number of CPU cores. As fully cached incremental pattern matching provides instantaneous match operations, it supports parallel transformation execution by allowing simultaneous access to the stateful pattern matcher from multiple model manipulation threads. By improving this scenario with concurrent maintenance phases, model manipulation (protected by concurrent programming mechanisms such as locks) will no longer force other transformation threads to wait for the termination of the time-consuming maintenance phase. As a consequence, read-intensive transformations are expected to scale well with parallel computational capacity. Since this thesis focuses on pattern matching, therefore the method of splitting the transformation into multiple model manipulation threads is not detailed here; see [4] for an elaboration of this approach including low-level implementation considerations.

4.1.2 Related work

Parallel graph transformations. In addition to large amount of theoretical work on concurrent and parallel aspects of graph transformation, relatively little practical work has been carried out. Some advanced solutions were proposed by G. Mezei [Mez07] who analyses pattern conflicts and groups executable rules into independence blocks to execute them in parallel. Further contributions also introduced parallel pattern search for first occurrence and all occurrences. My current work is complementary to his work, as it offers parallelization with a different pattern matching paradigm. Future research shall be conducted to identify how to combine the strength of the two approaches.

The GrGen.NET transformation system follows a different approach [Sch08] to avoiding parallelization conflicts: the model itself is split into several partitions, and different threads operate on different fragments. Pattern matches bridging multiple partitions are dealt with separately.

As an alternative solution to managing conflicts, Roberto Bruni has recommended\(^1\) investigating the possibility of employing an optimistic concurrency strategy in the future.

\(^1\)discussion during GT-VMT’09 in York, UK, March 2009
Parallel incremental pattern matching There is some work in literature in the context of parallel or distributed Rete implementations. For instance, [AT98] focuses on parallel rule applications, [MK90] aims at parallel pattern matching. Unfortunately, certain approaches focusing on expert systems are hard to be accessed, e.g. due to vague patent descriptions [Lin05], and certain industrial solutions might not be published at all. Anyhow, these approaches rarely provide proofs to guarantee the global termination of local updates as mentioned in Section 4.1.4, which is a requirement specific to the model transformation context.

The maintenance routine of the incremental matching approach TREAT [ML91], an alternative for Rete, invokes search-based pattern matching, therefore the parallel graph pattern matching techniques discussed earlier can be applied for it.

4.1.3 Concurrent pattern matching and model manipulation

The specific Rete net implementation used throughout Section 4.1 relies on asynchronous message passing, which is a distinct way of implementing the Rete maintenance routine \textit{MaintainRete}.

\textbf{Definition 62 (Rete maintenance with asynchronous message passing)} The asynchronous message passing version of Rete involves a \textit{message queue} data structure attached to the Rete net, containing \textit{update messages} that carry the match set delta of a parent node to a child node. An update message from node \(r_{\text{parent}}\) to \(r_{\text{child}}\) is \((r_{\text{child}}, \delta_{\text{parent}})\). There is a \textit{message consumption cycle} that loops fetching the first message \((r_{\text{child}}, \delta_{\text{parent}})\) from the queue and delivers it to the appropriate recipient \(r_{\text{child}}\), as long as there are any messages in the queue. Here delivery means that the \textit{Maintain}_{\text{child}}(\delta_{\text{parent}}) routine of the node will compute a delta match set \(\delta_{\text{child}}\); subsequently for each child \(r_{\text{grandchild}}\) of \(r_{\text{child}}\), a propagated output message \((r_{\text{grandchild}}, \delta_{\text{child}})\) will be placed to the end of the queue, thereby achieving asynchronous messaging. At the entry point of the matcher maintenance routine \textit{MaintainRete}, input node updates (derived from the model delta) are similarly put into the queue as messages. The rest of maintenance tasks simply consist of looping the message consumption cycle until the queue becomes empty.

Using asynchronous messaging in a multi-threaded context, the load on the main thread of the transformation can be reduced by executing the above message consumption cycle in a separate thread, as opposed to the model manipulation thread where the model was changed. When the transformation manipulates the model (see Figure 4.1(a)), it only has to send the new update message to the message queue, and can immediately resume its operation. The thread of the pattern matcher will execute the Rete cache maintenance in the background, ideally, without imposing a performance penalty on the transformation thread. When the message queue becomes empty, the Rete network has reached a \textit{fixpoint}; the pattern matcher thread then goes to sleep and will only resume its operation when a new update message is posted.

When the transformation issues a pattern query, it has to assure that all ongoing background message passing has terminated and the matches cached by Rete are up-to-date; so if necessary, it will have to wait (sleep) until Rete reaches its fixpoint.

\textbf{Example 25} Assuming that a fireable transition \(t\) was already identified in the Petri-net (using pattern \textit{fireable} from Example 16), Figure 4.1(b) shows how the Petri-net firing operation \textit{Fire}(t) (defined in Example 6) may behave in presence of concurrent Rete maintenance.

1. First, to determine the applicability of GT rule \textit{removeToken} (see Example 17) on the selected transition \(t\), matches of its its LHS pattern \textit{sourcePlace} is fetched instantaneously from the pattern matcher.
2. Then, applying the GT rule deletes one token in every source place, each of them issuing an update message to the pattern matcher thread that results in some asynchronous cache maintenance in the Rete net (e.g. to update the matches of the fireable pattern).

3. Next, after Rete maintenance is finished, the list of target places can retrieved from the pattern matcher.

4. Finally, a new token is created at each target place, resulting in subsequent notifications of the Rete maintenance thread.

Figure 4.1(c) displays the corresponding states of the Petri net model.

4.1.3.1 Initial performance results

While the local search based pattern matchers operate with cheap model changes and costly pattern queries, the sequential Rete-based matcher has a moderate overhead on model change balanced by instant pattern queries. This novel concurrent incremental pattern matching approach combines the advantages of the former two: it has cheap model manipulation costs, and potentially instant pattern queries. Although the transformation might have to wait for the termination of the background pattern matcher thread, the worst case of this time loss is still comparable to the update overhead of the original Rete approach.

This concurrent approach is expected to improve performance over a non-concurrent implementation (as described in Section 3.4.4) if there are comparatively infrequent pattern matcher queries and complex model changes between them. This would correspond to a “for all matches” style control flow when all matches of a pattern are obtained first, and then each of them is processed (potentially) simultaneously, which is common in model-to-model transformation scenarios. This complements the basic advantage of incremental pattern matching, which manifests especially on as long as possible
4.1. INCREMENTAL PATTERN MATCHING ON MULTI-CORE PLATFORMS

style control flows: when single matches are selected and processed one by one until there are no matches of the pattern.

Initial experiments carried out in 2009\textsuperscript{2} have shown that the concurrent approach improves performance by up to 20% on the Sierpinsky benchmark of [The07]. For building a Sierpinsky-triangle of 8, 9 and 10 generations, the single-threaded Rete-based solution ran for 2.6s, 8.3s, and 26.2s, while the concurrent solution took 2.2s, 6.9s, 22.8s to terminate, respectively, with an average performance gain of 18%.

4.1.4 Multi-threaded pattern matching with Rete

The concurrent patten matching approach can be improved further given that the hardware architecture is capable of running multiple threads efficiently. There are various approaches of parallelizing the Rete algorithm, see Section 4.1.2 for details. Here a simple solution is presented.

The basic idea is to employ multiple pattern matcher threads to consume update messages. However, if these threads share the same message queue and Rete nodes, and multiple threads could access the same node simultaneously, this could easily lead to complex inconsistency problems, which could not be easily avoided by locks.

The proposed solution splits the network into separate Rete containers. Each Rete container has its own distinct set of Rete nodes with a separate message queue, and is assigned to a dedicated pattern matcher thread running the message consumption cycle on the corresponding queue. Each container is responsible for delivering messages to its nodes using its message queue. This way, no two threads are allowed to operate on the same Rete node, thus implementing node-level mutual exclusion is not necessary. Forwarding messages between two containers is accomplished by enqueuing the message in the target container.

Example 26 Figure 4.2(a) depicts a parallel version version of the Rete matcher in Example 21, illustrating how a Rete net can be split into several containers for parallel execution.

---

\textsuperscript{2}Environment: 2.2GHz Intel Core 2 Duo processor, Windows Vista, Sun Java 1.6.0_11, 1GB heap memory
Example 27 Figure 4.2(b) illustrates how changes performed by the transformation induce propagation of update messages in the Rete net. The update messages spread between containers, and are processed in parallel. When the transformation needs to match a pattern, however, it will have to synchronize with the pattern matcher and wait until all Rete activities have settled.

Definition 63 (Fixpoint states of multi-container Rete net) If a Rete container runs out of update messages to process, it reaches a local fixpoint, otherwise it remains active. The global fixpoint is reached when all containers are in a local fixpoint.

In order to retrieve up-to-date and consistent match sets, the transformation thread has to wait for a global fixpoint. This thread synchronisation goal, however, is not trivial to accomplish, since a container can leave its local fixpoint and become active again before a global fixpoint is reached due to incoming messages from other, still active containers. To address this issue, a termination protocol based on logical clocks has been developed and implemented.

4.1.4.1 Termination protocol for Rete containers (algorithm sketch)

Each container $C_i$ is equipped with a logical clock (denoted as $\text{clock}_i$) that is incremented whenever a local fixpoint is reached by the message consumption thread of the container (denoted $\text{thread}_i$). Each time container $C_i$ sends an update message to container $C_k$, the message is appended to the message queue of $C_k$ and the value of $\text{clock}_k$ is retrieved and stored in $C_i$ as $\text{criterion}_i[k]$, all as a single atomic step\(^3\). The retrieved clock value imposes a termination criterion: the network can only reach a global fixpoint if the value of $\text{clock}_k$ exceeds the received snapshot. This means that the relayed message has been delivered to the recipient node in $C_k$ and all of the (local) consequences have been resolved, resulting in a new local fixpoint in $C_k$, which is required for a global fixpoint.

When $C_i$ reaches its local fixpoint, it atomically increments its clock and reports the event to a global Rete network object; this report includes the incremented $\text{clock}_i$, along with the values $\text{criterion}_i[k]$ for each $k$. Similarly, when the transformation changes the model and consequently sends a change notification (formulated as an update message) to an input node in container $C_k$, it hands over the message to the message queue of $C_k$, fetches the $\text{clock}_k$ from that container and reports it to the network object as a termination criterion, also performed as a single atomic step.

The global network object maintains an array $\text{criterion}_{\text{global}}[k]$ storing the largest reported criterion for each $k$, and $\text{clock}_{\text{reported}}[i]$ for the latest clock value reported by $C_i$. Upon receiving the report, the global network object evaluates whether a global fixpoint is reached and wakes the transformation thread when appropriate. Determining whether a global fixpoint holds is as simple as checking, for each container, whether the highest reported termination criterion value stemming from that container is exceeded by its the latest reported fixpoint-time clock value. This Termination Condition is formulated as:

$$\forall k : \text{clock}_{\text{reported}}[k] > \text{criterion}_{\text{global}}[k]$$  \hspace{1cm} (4.1)

4.1.4.2 Proof of correctness and liveness

I will formally prove that the outlined termination algorithm is correct and deadlock-free. Correctness refers to the design goal that the Termination Condition (Equation 4.1) holds exactly when the system is in a global fixpoint. The deadlock-free property is valid for the maintenance (message consumption)

\(^3\)The outlined procedure is only necessary if $k \neq i$; messages sent and received within the same container can use the message queue without any interaction with clocks
threads, as well as for the model manipulation thread that can possibly wait for a global fixpoint to match a pattern.

**Theorem 3** During the execution of the Termination protocol in Section 4.1.4.1, the Termination Condition (Equation 4.1) holds exactly when the distributed Rete net is in a global fixpoint.

**Proof (Termination Condition \(\implies\) all containers are in a local fixpoint)** Proof by contradiction: let us assume that some containers are not in a fixpoint. From those containers, select the one that has been active (not in a fixpoint) for the longest time (this time interval is finite, as all containers initialise in a fixpoint); name that container \(C_k\). Termination Condition (Equation 4.1) implies that \(\text{clock}_k = \text{clock}_{\text{reported}}[k] > \text{criterion}_{\text{global}}[k]\), meaning that a termination criterion of value \(\text{clock}_k\) was never reported for \(C_k\), since \(\text{criterion}_{\text{global}}[k]\) can only increase.

Whichever message marked the end of the last fixpoint of \(C_k\), its sender received the value \(\text{clock}_k\) as a termination criterion; this criteria was never reported. Had the sender been a transformation, the criteria would have been atomically reported to the network according to the protocol; this means that the sender must have been a container \(C_i\). Since \(C_k\) is asserted to have been active longer than \(C_i\), \(C_i\) must have reached a local fixpoint since sending that message, and then it should have reported \(\text{criterion}_i[k] = \text{clock}_k\) to the global network object. This is a contradiction.

**Proof (Termination Condition \(\iff\) all containers are in a local fixpoint)** Proof by contradiction: let’s assume that some \(k\) violates the Termination Condition (Equation 4.1), i.e. \(\text{clock}_k = \text{clock}_{\text{reported}}[k] = \text{criterion}_{\text{global}}[k]\). The criteria was reported either by a transformation or by another container. In the former case, the transformation was sending a message to \(C_k\) when delivering this report; in the latter case, a container \(C_i\) delivered a message to \(C_k\) before reporting. In either case, at some point in the past, \(C_k\) received a message when \(\text{clock}_k\) already had its present value. Even if it had been in a local fixpoint, receiving the message made \(C_k\) active. Since according to the assumption, \(C_k\) is in a fixpoint at the present time, it must have reached a new fixpoint since the message was received. This means that \(\text{clock}_k\) must have been incremented since the message was received, but \(\text{clock}_k = \text{clock}_{\text{reported}}[k]\) is still the same, which is a contradiction.

**Theorem 4** The execution of the Termination protocol in Section 4.1.4.1 will not halt at a deadlock before reaching the global fixpoint.

**Proof (The protocol cannot halt in a deadlock)** The critical section of the global network object is only entered for short reporting routines, during which there is no waiting, so the critical section will eventually be free again. This also means that waiting to enter this critical section will not cause a deadlock. A thread can, however, suspend execution and wait within critical sections of a message queue, but the only way this can happen is while trying to enter the inner critical section. As the inner critical section belongs to the global network object and is already shown to be live, critical sections of a message queue can also be freed.

The only remaining way a message consumption thread can be forced to wait is when it is in a local fixpoint and waiting for a new message. In this case, if there are active containers remaining, they can operate further. Finally, if all message consumption threads are waiting, then all containers are in a local fixpoint, so global fixpoint is reached, and the transformation thread is not forced to wait.
4.1.4.3 Performance expectations

It is important to point out that the performance of such a system may depend highly on the amount of synchronization and replication that is necessary when messages are passed between the containers. If only one of the containers is active most of the time, there is little performance gain. Even worse, if there is continuous message passing between the containers, the overhead may even reduce performance. Using a naive strategy to distribute the nodes of Rete nets, I was not able to observe significant performance improvements in any of my experiments.

In theory, it would seem beneficial if subpatterns deployed to separate Rete containers had a low number of interconnections, but further research is necessitated to achieve this in practice. An ideal application scenario would be several transformations or parts of the same transformation that are known to use different patterns; allowing easy, straightforward splitting and parallelization of the Rete net, with a low amount of inter-connectedness. By partitioning the patterns into relatively independent containers, a multi-threaded Rete pattern matcher may achieve high performance. Investigating such partitioning strategies requires future research.

4.2 Graph patterns with transitive closure

This section partially follows [8].

4.2.1 Introduction

In modeling scenarios, transitive closure is generally needed to express model properties which are recursively defined, often used in reasoning about partial orders, and thus widely found in modeling applications, e.g. to compute model partitions or reachability regions in traceability model management [AKM+10] and business process model analysis [KGV08]. In graph transformations, recursive graph patterns are most frequently used to specify transitive closure for processing recursive model structures [VB07]. At the meta-level, they may provide the underpinnings for n-level metamodeling hierarchies where transitive type-subtype-instance relationships need to be maintained [MLM08], or for maintaining order structures such as those proposed in [3] for spatially aware stochastic simulation.

Incremental graph pattern matching has already demonstrated scalability in a number of such scenarios [2,16,11], especially when pattern matching operations are dominating at runtime (e.g. view maintenance, model synchronization, well-formedness checking and state space traversal [GJR10]). However, recursive incremental pattern matching is supported only for acyclic subgraphs (see Section 2.2.2.3). Therefore, as it has been recently recognized in [JRG12], the efficient integration of transitive closure computation algorithms for graphs would provide a crucial extension to the current capabilities of incremental pattern matchers.

In order to efficiently adapt transitive closure computation for the specific needs of incremental graph pattern matching, three key challenges need to be addressed.

• First, cyclic closure is not addressed by the presented pattern matching solution so far. The Rete algorithm does not handle recursive graph pattern composition correctly (which is otherwise out of scope for the thesis, see Section 2.2.2.3), i.e. in the presence of cycles in the graph, incremental updates of recursive patterns may yield false matching results. Even if initial computations ($s_0$) are correct, incremental updates may be incorrect. This means that transitive closure, while expressible as a recursive graph pattern, can not in general be provided by Rete (see Definition 4.2.3 on page 84).
4.2. GRAPH PATTERNS WITH TRANSITIVE CLOSURE

• Second, for functionally complete pattern matching it is important to support generic transitive closure, i.e. the ability to compute the closure of not only simple graph relations (features), but also derived features defined by binary graph patterns that establish a complex logical link between a source and a target vertex.

• Finally, the adaptation should align with the general performance characteristics of incremental pattern matching and impose a low computational overhead on model manipulation operations and minimize runtime memory overhead.

To address the above challenges, a generic transitive closure extension is introduced to the graph pattern formalism. To facilitate the needs of incremental graph pattern matching, several general purpose incremental graph transitive closure algorithms [PvL88, FMNZ00] were investigated. After analyzing common characteristics of several modeling scenarios, I co-developed IncSCC, a novel version of an incremental transitive computation algorithm [PvL88] based on the maintenance of strongly connected components.

Section 4.2.2 formally introduces concepts related to transitive closure and provides a language extension to the graph pattern formalism. Section 4.2.3 elaborates the Rete-based approach for the extended language. Section 4.2.4 presents the general purpose incremental graph transitive closure algorithms that were adapted in the solution.

Further details of the approach, description of the implementation in Viatra2, and case study-driven performance evaluation reports are available in [8].

4.2.2 The transitive closure problem

4.2.2.1 The notion of transitive closure

Definition 64 (Transitive closure of a binary relationship) For a binary relationship \( E \subseteq D \times D \) over a domain \( D \subseteq \mathbb{U} \), the irreflexive transitive closure \( E^+ \subseteq D \times D \) consists of \( \langle u, v \rangle \in D \times D \) pairs of elements for which there is a non-empty finite linked sequence \( \langle w_0, w_1 \rangle, \langle w_1, w_2 \rangle, \ldots, \langle w_{k-1}, w_k \rangle \) of pairs where \( \langle w_i, w_{i+1} \rangle \in E \) for each pair in the sequence and \( u = w_0 \) and \( w_k = v \). On the other hand, the reflexive transitive closure \( E^* \subseteq D \times D \) consists of \( E^+ \cup \{ \langle v, v \rangle \mid v \in D \} \).

In case of generic transitive closure, the base relationship is a “derived feature”, not restricted to simple graph edges (relations/features in our graph model terminology), but defined by any two-parameter graph pattern (e.g. with path expressions, attribute checks). We focus on the most general approach: generic, irreflexive transitive closure.

Definition 65 (Binary pattern) A binary pattern is a graph pattern \( P(2) = \langle V, C \rangle \in \text{Patterns}_{MM}^{(2)} \) with two (parameter) variables \( V = v_{\text{from}}, v_{\text{to}} \) where the order of the (parameter) variables is well-defined. The set of all binary patterns over a metamodel \( MM \) is \( \text{Patterns}_{MM}^{(2)} \).

Using the tuple notation for matches and observing the variable order associated with the binary pattern, the match set \( \text{MatchSet}_{G}^{P(2)} \) of the binary pattern in a graph model \( G : MM \) can be represented as a binary relationship: \( \text{MatchSet}_{G}^{P(2)} \subseteq \text{Elements}_G \times \text{Elements}_G \).

Definition 66 (Irreflexive transitive closure of binary pattern) For binary pattern \( P(2) \in \text{Patterns}_{MM}^{(2)} \) and graph model \( G \in \text{Graphs}_{MM} \) over the same metamodel \( MM \), the irreflexive transitive closure \( P_{(2)}^+ \) of the pattern is defined as the irreflexive transitive closure \( (\text{MatchSet}_{G}^{P(2)})^+ \).
Definition 67 (Transitive closure pattern constraint) In a graph pattern \( P = (V, C) \) over meta-model \( MM \), (irreflexive) transitive closure constraints \( C^{tc} \subseteq V^{ent} \times V^{ent} \times \text{Patterns}_{MM}^{(2)} \) state that the given pair of variables represent a pair in the irreflexive transitive closure of the given binary pattern. A substitution \( s \) satisfies a transitive closure constraint \( c = \langle u, v, P_{(2)} \rangle \in C^{tc} \) iff \( \langle s(u), s(v) \rangle \in P_{(2)}^+ \). \( \text{Dom}(c) = \{ u, v \} \).

Example 28 Using the structural metamodel of Example 2 on page 11, the fact that the firing of a given Transition may potentially enable another one is expressed by the binary, disjunctive graph pattern \( P_{\text{mayEnable}} = \langle \{ \text{from}, to \}, \{ P_{\text{give}}, P_{\text{take}} \} \rangle \) with parameter variables \( \text{from}, to \) and body patterns \( P_{\text{give}} \) and \( P_{\text{take}} \). Here \( P_{\text{give}} = \langle \{ \text{from}, \text{outArc}, \text{place}, \text{inArc}, to \} \rangle, \{ c_1, c_2 \} \rangle \) expresses that firing the transition \( \text{from} \) may put a token in place \( \text{place} \) and thus enable transition \( \text{to} \), with \( c_1 = \langle \text{from}, \text{outArc}, \text{place}, \text{OutArc} \rangle \in C^{rel} \) and \( c_2 = \langle \text{place}, \text{inArc}, \text{to}, \text{InArc} \rangle \in C^{rel} \). Also, \( P_{\text{take}} = \langle \{ \text{from}, \text{inArc}, \text{place}, \text{inhibit}, to \} \rangle, \{ c_3, c_4 \} \rangle \) expresses that firing the transition \( \text{from} \) may remove a token from place \( \text{place} \) and thus stop the inhibition of transition \( \text{to} \), with \( c_3 = \langle \text{place}, \text{inArc}, \text{from}, \text{IncArc} \rangle \in C^{rel} \) and \( c_2 = \langle \text{place}, \text{inhibit}, to, \text{InArc} \rangle \in C^{rel} \). Now the transitive closure of this binary pattern (with ordering convention \( \text{from} \to \text{to} \)) would express a primitive approximation of what transitions may potentially become enabled after a sequence of firings starting with a selected transition. In a model checker or other state space exploration scenario, this transitive closure may be useful in a NAC subpattern, to reduce the search space by avoiding transitions that are provably not going to help in enabling a certain goal transition.

Note that a reflexive version of transitive closure pattern constraint can be defined similarly, with one important difference: reflexive closure cannot be computed from the base relationship \( E \) alone, without knowledge of the domain \( D \). Thus the reflexive transitive closure is not well-defined for binary patterns; it is usable with e.g. relation types, however. From an implementation point of view, the reflexive closure is trivial to derive from the irreflexive one by adding \( \{ \langle d, d \rangle \mid d \in D \} \), therefore the thesis will focus on the latter case.

4.2.2.2 Transitive closure in graph theory

The following important concepts will be related to transitive reachability in digraphs of the graph theory sense (not to be confused with the more complex notion of graph model, see Section 2.1).

Definition 68 (Directed graph (digraph)) A digraph \( G = (V, E) \) is a set of vertices \( V \subset U \) and a binary relation \( E \subseteq V \times V \), also called the set of edges.

Definition 69 (Strongly connected component (SCC)) A digraph \( (V, E) \) is strongly connected iff all pairs of its vertices \( u, v \in V \) are mutually transitively reachable (\( (u, v) \in E^+ \)). An SCC of a digraph is a maximal subset of vertices \( s_i \subseteq V \) within the digraph that is strongly connected.

As the SCC of a vertex \( v \) is the intersection of the set of ancestors and descendants of the vertex, each digraph \( G \) has a unique decomposition \( S_G \) into disjoint SCCs.

Definition 70 (Condensed graph) For a digraph \( G = (V, E) \), the SCCs form a digraph \( G_c(S_G, E_c) \) called condensed graph. The condensed graph has the SCCs as its vertices, where two SCCs are connected iff any of their vertices are connected in the original digraph: \( E_c = \{ \langle s_i, s_j \rangle \mid s_i, s_j \in S_G \land \exists u \in s_i, v \in s_j : \langle u, v \rangle \in E \} \).

It follows from the definitions that a condensed graph is always acyclic.
4.2.2.3 Stateful computation of transitive closure

In analogy to stateful pattern matchers (see Section 3.1), I now introduce the notion of stateful transitive closure computation. The algorithm state is initially constructed from a binary base relationship; this state is a data structure for storing the result and possibly auxiliary information as well. The algorithm can subsequently answer transitive reachability queries (e.g. which elements are reachable from a given element, etc.) efficiently, using information cached in the state. When the base relationship is changed by a delta, the stored state can be maintained, and a delta of the transitive closure of the base relationship is calculated.

A stateful transitive closure algorithm \( CL = (S_{CL}, Construct_{CL}, Query_{CL}, Maintain_{CL}) \) over a domain \( D \) consists of:

- a set of all permitted states \( S_{CL} \);
- \( Construct_{CL} : 2^{D \times D} \rightarrow S_{CL} \) a function that maps a binary relationship \( E \subseteq D \times D \) to the state of the algorithm corresponding to \( E \);
- \( Query_{CL}(Src, Trg) : S_{CL} \rightarrow 2^{D \times D} \) is a function that extracts a binary relationship (the transitive closure or a selected reduced subset) from the state, parameterized by \( Src, Trg \in D \cup \{\ast\} \), so that for relationship \( E \) and algorithm state \( s_{CL} \in S_{CL} \) with \( s_{CL} = Construct_{CL}(E) \), \( s_{CL}.Query_{CL}(Src, Trg) = \{ \langle u, v \rangle | \langle u, v \rangle \in E^+ \land (Src \neq \ast \Rightarrow u = Src) \land (Trg \neq \ast \Rightarrow v = Trg) \}; \)
- \( Maintain_{CL}(\Delta E) : S_{CL} \rightarrow (S_{CL} \times \{-1, 0, +1\}^{D \times D}) \) is a function that, upon a change \( \Delta E \) of base relationship \( E \), updates algorithm state \( s_{CL} \) to \( s'_{CL} = s_{CL}.Maintain_{CL}(\Delta E) \), where \( s'_{CL} = Construct_{CL}(E + \Delta E) \), and returns the delta \( s_{CL}.Maintain_{CL}(\Delta E).\delta = (E + \Delta E)^+ - E^+ \).

The maintenance routine of such a stateful algorithm is often more conveniently defined for single-element deltas. If \( s_{CL}.Insert_{CL}(Src, Trg) \) maintains transitive closure after the insertion of \( \langle Src, Trg \rangle \) into \( E \) and \( s_{CL}.Delete_{CL}(Src, Trg) \) is the maintenance routine after the deletion of \( \langle Src, Trg \rangle \) from \( E \), then \( s_{CL}.Maintain_{CL}(\Delta E) \) can be decomposed into sequential invocations of \( Insert_{CL} \) and \( Delete_{CL} \) for each element of \( \Delta E \).

Analogously to matchers, a stateful transitive closure algorithm is incremental if \( Maintain \) computes the delta of the algorithm state efficiently from the delta of the base relationship. Note that the word “incremental” has a more restrictive meaning in the transitive closure algorithm research community, therefore this case is frequently called in literature the (fully) dynamic transitive closure problem.

4.2.3 Integration of transitive closure into Rete

Following [JRG12], a transitive closure result will be represented by a Rete node. Dynamic (incremental) transitive closure algorithms are integrated into Rete nodes by exploiting the operations specified in Section 4.2.2.3. Generic transitive closure (see Section 4.2.2.1) is achieved by attaching such a Rete node to a parent node that matches an arbitrary binary graph pattern (derived feature).

The transitive closure Rete node is an operation node which receives updates from a parent node representing a binary graph pattern \( P(2) \) and forms a two-way interface between Rete and an incremental transitive closure maintenance algorithm \( CL \). Whenever the Rete node for \( P(2)^+ \) has to be
maintained according to the delta from its parent node \(v(P_{(2)})\). \(Maintain_{CL}\) is invoked. The subroutine computes the necessary updates to \(P_{(2)}^+\), and returns these delta pairs, which will then be propagated to child nodes.

**Definition 71 (Transitive closure node)** For transitive closure constraint \(c = \langle u, v, P_{(2)} \rangle \in C^{tc}\), the transitive closure node associated with stateful transitive closure algorithm \(CL = \langle S_{CL}, Construct_{CL}, Query_{CL}, Maintain_{CL} \rangle\) is a type of operational node \(r_c = \langle P_c, Parents_c, M^0_c, Maintain_c \rangle\), where

- **Parents**: there is a single parent \(Parents_c = \{v(P_{(2)})\}\),
- **Pattern**: \(P_c = \langle V, C \rangle\) with \(V = \{u, v\}\) and \(C = \{c\}\),
- **Memory**: the memory stores the algorithm state \(M_c = s_{CL}\) with \(M_c.matches = s_{CL}.Query_{CL}(\ast, \ast)\),
- **Invariant**: the consistency invariant is \(M_c.matches = \text{MatchSet}_{P_c} = P_{(2)}^+\)
- **Initialization**: \(M^0_c = Construct_{CL}(\emptyset)\) with \(M^0_c.matches = \emptyset^+ = \emptyset\)
- **Maintenance**: \(M_c.Maintain_c(M_{(P_{(2)})}.matches \oplus \Delta_{v(P_{(2)})})\) is defined in the following, using notation \(M_{(2)} = M_{(P_{(2)})}.matches, \Delta_{(2)} = \Delta_{v(P_{(2)})}.matches\), by specifying the new memory and the calculated delta: \(M'_c = M_c.Maintain_{CL}(\Delta_{(2)}).r\) and \(\Delta_c.matches = M'_c.matches - M_c.matches = (M_{(2)} + \Delta_{(2)})^+ - M_{(2)}^+ = M_c.Maintain_{CL}(\Delta_{(2)}).\delta\).

It follows easily from the definitions of \(M^0_c\) and \(Maintain_c\) and additionally the properties of \(CL\) that the consistency invariant is preserved, and that the maintenance computation is incremental.

Once the transitive closure node is constructed for a transitive closure pattern constraint, it can be used as a leaf node in the join tree along with input nodes.

In practice, \(Construct_{CL}\) may be used as a shortcut for initializing the transitive closure node. In addition to obtaining \(M_c.matches\) when initializing the child nodes, transitive reachability queries may be invoked as a quick lookup to speed up join operations on the node contents.

As an alternative approach, transitive closure can be expressed as a recursive graph pattern (see Section 2.2.2.3 on page 29). This solution was rejected as Rete has first-order semantics without fixpoint operators, and might therefore incorrectly yield a (still transitive) superset of the transitive closure when attempting to match such a recursive pattern. This would mean that in graph models containing cycles (or taking the closure of a derived feature that is otherwise cyclic), when such cycles are broken by model manipulation, obsolete reachabilities could cyclically justify each other after their original justification was deleted. Note that on the other hand, if the derived feature is acyclic, or pairs in the binary relationship are never removed, then the results are correct. Similarly, the results of the initial computation are also correct (as no relations have been removed yet).

### 4.2.4 Incremental graph transitive closure maintenance algorithms

An incremental transitive closure algorithm is required to operate the Rete node proposed in Section 4.2.3. From the rich literature (see Section 4.2.4.1), two such algorithms were selected and adapted. An overview of their core ideas is provided in Section 4.2.4.2 and Section 4.2.4.3.
4.2. GRAPH PATTERNS WITH TRANSITIVE CLOSURE

4.2.4.1 State of the art

While there are several classical algorithms (depth-first search, etc.) for computing transitive reachability in digraphs, efficient incremental maintenance of transitive closure is a more challenging task. As transitive closure can be defined as a recursive Datalog query, incremental Datalog view maintenance algorithms such as DRed [GMS93] can be applied as a generic solution. There is also a wide variety [DI06] of algorithms that are specifically tailored for the fully dynamic transitive reachability problem. Some of these algorithms provide additional information (shortest path, transitive reduction), others may be randomized algorithms (typically with one-sided error); the majority focuses on worst-case characteristics on dense digraphs. The spectrum of solutions offers various trade-offs between the cost of operations specified in Section 4.2.2.3.

Even if the original digraph has a moderate amount of edges (sparse graph), the size of the transitive closure relationship can easily be a quadratic function of the number of vertices, raising the relative cost of maintenance. A key observation, however, is that in many typical cases of software engineering models, vertices will form large SCCs. This is exploited in a family of algorithms [PvL88, FMNZ00] that maintain (a) the set of SCCs using a dynamic algorithm, and also (b) the transitive reachability relationship between SCCs. Choosing such an algorithm is justified by simplicity of implementation, the sparse property of typical graph models and the practical observation that large SCCs tend to form.

4.2.4.2 DRed - Delete and REDerive

DRed [GMS93] is a general-purpose approach for incremental result maintenance of recursive Datalog queries, applied here specifically for the transitive closure problem.

The dynamic transitive closure algorithm \( \text{DRed} = (\text{S}_{\text{DRed}}, \text{Construct}_{\text{DRed}}, \text{Query}_{\text{DRed}}, \text{Maintain}_{\text{DRed}}) \) explicitly stores the transitive closure, i.e. \( s_{\text{DRed}} = E^+ \). \text{Construct}_{\text{DRed}} is therefore simply a non-incremental transitive closure computation, using any known algorithm for the static problem. \text{Query}_{\text{DRed}} is directly answered based on \( E^+ \). The update operations are derived from the DRed algorithm for recursive Datalog queries.

\( \text{Insert}_{\text{DRed}}(\text{Src}, \text{Trg}) \) computes the newly reachable pairs as \( E^* \circ \{ (\text{Src}, \text{Trg}) \} \circ E^* \), and adds them to \( E^+ \) (unless already reachable), where \( A \circ B := \{ (u, v) | \exists w : (u, w) \in A \land (w, v) \in B \} \).

\( \text{Delete}_{\text{DRed}}(\text{Src}, \text{Trg}) \) computes an overestimation of the delta as \( E^+_D = (E^* \circ \{ (\text{Src}, \text{Trg}) \} \circ E^*) \setminus E \), and marks these pairs for deletion. Then it attempts to iteratively derive again these marked reachability pairs using unaffected ones as \( E^+_D \cap (E \circ (E^+ \setminus E^+_D)) \); successfully rederived pairs are removed from \( E^+_D \), allowing further ones to be rederived until a fixpoint is reached. The final contents of \( E^+_D \) are the deleted reachability pairs that should be removed from \( E^+ \).

4.2.4.3 IncSCC - Incremental Maintenance of Strongly Connected Components

\( \text{IncSCC} = (\text{S}_{\text{IncSCC}}, \text{Construct}_{\text{IncSCC}}, \text{Query}_{\text{IncSCC}}, \text{Maintain}_{\text{IncSCC}}) \) is a fully dynamic transitive closure algorithm, the adaptation of an original algorithm from [PvL88]. In the following, a brief overview is given of IncSCC. For details and analysis, refer to [PvL88].

The main idea is to reduce update time and memory usage by eliminating unnecessary reachability information. The fact that each vertex is reachable from every other vertex within the same SCC is identified as unnecessary to store. Thus, instead of storing the entire transitive closure relationship, the two concerns of the algorithm are maintaining (i) a decomposition into SCCs, and (ii) transitive reachability within the condensed graph. The latter is a simpler problem with several efficient solutions, as the condensed graph is acyclic; the presented implementation relies on
the “basic algorithm” from the original paper [PvL88], that will be called the Counting Algorithm Counting = \( (S_{\text{Counting}}, \text{Construct}_{\text{Counting}}, \text{Query}_{\text{Counting}}, \text{Maintain}_{\text{Counting}}) \), as it simply keeps track of the number of derivations for each transitive reachability pair.

As the most significant improvement over [PvL88], the transitive closure relation \( E^+ \) is not stored explicitly in IncSCC. The state \( s_{\text{IncSCC}} \) maintained by the algorithm consist of:

- a decomposition \( s_{\text{IncSCC}}.S_G \) of the digraph \( G \) into SCCs, and
- algorithm state \( s_{\text{IncSCC}}.s_{\text{Counting}} \) for incremental computation of transitive reachability within the condensed graph \( G_c \) using the Counting Algorithm.

Construct\(_{\text{IncSCC}}\)(\( E \)): The SCC partitioning of the initial graph is computed using Tarjan’s algorithm [Tar72] based on depth-first search. Afterwards, the condensed graph is constructed, and the Counting Algorithm is initialized to provide reachability information between SCCs.

Query\(_{\text{IncSCC}}\)(\( \text{Src}, \text{Trg} \)): \( E^+ \) in digraph \( G(V, E) \) can be reconstructed from the partitioning \( S_G \) of SCCs and the reachability relation \( E^+_c \) of condensed graph \( G_c(S, E_c) \), since for \( s_1, s_2 \in S_G, s_1 \neq s_2, u \in s_1, v \in s_2 : (s_1, s_2) \in E^+_c \iff (u, v) \in E^+ \). Therefore when receiving a reachability query Query\(_{\text{IncSCC}}\)(\( \text{Src}, \text{Trg} \)), the parameter vertices \( \text{Src} \) and \( \text{Trg} \) are mapped to SCCs (unless they are \( \ast \)); reachability information in the condensed graph is provided by the Counting Algorithm. Vertices enumerated in the answer are obtained by tracing back the SCCs to vertices.

For example, \( s_{\text{IncSCC}}.\text{Query}_{\text{IncSCC}}(\text{Src}, \ast) \) is answered by \( s_{\text{IncSCC}}.S_G^{-1}(s_{\text{IncSCC}}.s_{\text{Counting}}.\text{Query}_{\text{Counting}}(s_{\text{IncSCC}}.S_G(\text{Src}), \ast)) \).

Insert\(_{\text{IncSCC}}\)(\( \text{Src}, \text{Trg} \)): First, a lookup in \( s_{\text{IncSCC}}.S_G \) maps the vertices to SCCs. Afterwards, there are three possible cases to distinguish.

- If \( \langle \text{Src}, \text{Trg} \rangle \) are in different SCCs, the new edge of the condensed graph is handled by the Counting Algorithm, which can confirm that no cycle is created in the condensed graph.
- If, however, the inserted edge caused a cycle in the condensed graph, then the cycle is collapsed into a single SCC.
- Finally, if (iii) \( \langle \text{Src}, \text{Trg} \rangle \) are in the same SCC, there is no required action.

Delete\(_{\text{IncSCC}}\)(\( \text{Src}, \text{Trg} \)): The algorithm first performs a lookup in \( s_{\text{IncSCC}}.S_G \) to map the vertices to SCCs; afterwards, we once again distinguish three possible cases.

- If \( \langle \text{Src}, \text{Trg} \rangle \) are in the same SCC but \( \text{Trg} \) remains reachable from \( \text{Src} \) after the edge deletion (as confirmed by a depth-first-search), no further actions are required.
- If \( \langle \text{Src}, \text{Trg} \rangle \) are in the same SCC but \( \text{Trg} \) is no longer reachable from \( \text{Src} \) after the edge deletion, then the SCC is broken up (using Tarjan’s algorithm) into smaller SCCs, because it is no longer strongly connected.
- Finally, if \( \langle \text{Src}, \text{Trg} \rangle \) are in different SCCs, then the edge is deleted from the condensed graph, which is in turn is handled by the Counting Algorithm.
4.2.5 Related work

Apart from ViATRA2, GROOVE [GJR10] also features a Rete-based incremental pattern matcher, and is therefore the most closely related work. In fact, the Rete implementation in GROOVE has recently been extended [JRG12] by the capability of incrementally maintaining transitive closure relations. The solution presented above is based on their idea of introducing a new type of Rete node that accepts a binary relationship as input and emits its transitive closure as output. The transitive closure node in GROOVE implements a simple algorithm that maintains the set of all paths (walks) of any length that can be composed from the original binary relationship, even if many of those paths are redundant due to having the same sources and targets. This results in factorial time and space complexity, as opposed to the various polynomial solutions found in literature and also in the approach presented here. Furthermore, their solution is only capable of computing the transitive closures of so-called regular (path) expressions; the notion of “derived feature” as presented here is more general, as it includes arbitrary graph structures (e.g., circular patterns as context, attribute restrictions, etc.). Finally, the experimental assessment in [JRG12] is conducted under special conditions, such as the graph being linear; in contrast, [8] finds the ViATRA2 implementation of the presented approach to be scalable on a graph structure without such restrictions (P2P network modeled as graph with branches and cycles).

In the future, it would be interesting to carry out experimental comparison of the transitive closure features of GROOVE and ViATRA2. This will need significant additional effort, as the running example of [8] relies on a complex peer-to-peer model and a stochastic simulator engine that would be difficult to replicate on GROOVE, while the case study example in [JRG12] relies on model checking capabilities that are not supported in ViATRA2.

Some other graph transformation tools [Sch90, NNZ00] feature path expressions, including transitive closure, without maintaining the result incrementally. In a graph with a low branching factor, they can still be a feasible alternative of incremental approaches in practice.

There are other model transformation tools that offer incremental evaluation. The incremental transformation solution in ATL [JT10], among other approaches, relies on impact analysis of OCL [OMG12a] expressions, meaning that the entire OCL expression will be re-evaluated whenever a relevant element in the model changes; however, standard OCL can only very recently express transitive closure in arbitrary graphs. There is an incremental evaluation technique for Tefkat [HLR06] that maintains an SLD resolution tree of the pattern expression; but without special handling of transitive closure, the SLD tree expands all possible paths from source to target, leading to factorial complexity similarly to GROOVE.

4.3 Incrementality on top of existing relational databases

This section partially follows [7].

4.3.1 Motivation

As industrial practice demands ever larger system models, the scalability of storage, query and manipulation of complex graph-based model structures gains importance, including the efficient execution graph transformation, and graph pattern matching in particular. For industrial applications, compatibility and integration with already well-established technologies is preferred to custom solutions. Relational Databases (RDBs) have successfully served as the storage medium for business critical data
88	CHAPTER 4. ADVANCED INCREMENTAL PATTERN MATCHING

for large companies. As explored in [VFV05], RDBs offer a promising implementation environment for large graph models and graph transformation.

Regarding execution performance of graph transformation, however, RDBs have had mixed success [VSV05a]. Graph transformation with incremental pattern matching in RDBs has been proposed in [VV04]. However, this approach guarantees the consistency of incremental caches only if the model is restricted to evolve along the specified GT rules. Therefore this solution is not compatible with already deployed (legacy) software, which may manipulate the underlying database in an arbitrary way. In fact, in many industrial scenarios, the underlying relational database (where the graph model is stored) is accessed in multiple ways (client programs, server side scripts), which are unaware of the incremental caches, hence they do not update them properly. For consistent behavior, these programs would have to be re-engineered with high effort.

I propose to extend existing solutions of incremental pattern matching over RDBs in order to obtain an efficient system in an industrial environment that can complement already deployed software. With the presented approach, incrementality will be guaranteed regardless of any external changes to the underlying database.

**Goals** To summarize, the proposed solution will keep the beneficial properties of [VV04], including Declarativity (automatic execution based on GT specification, without requiring manually written code) and Incrementality (incremental evaluation techniques for the graph pattern matching phase, to improve performance on certain types of tasks). Additionally, the new requirement of Compatibility will also be addressed, permitting side-by-side operation with any existing legacy scripts and programs already reading or writing the database contents.

Section 4.3.2 will give an overview of the approach that is proposed to meet these goals, the key component of which is the incremental pattern matcher. Before elaborating the details of the solution, Section 4.3.3 will describe a method from [VFV05] for (stateless) pattern matching in relational databases. This known technique is then extended in Section 4.3.3 to achieve the remaining goal of Incrementality (while observing Compatibility) in case of the basic graph pattern formalism. Finally, Section 4.3.5 discusses advanced pattern language elements.


**4.3.2 Overview of the approach**

The presented novel approach aims to conduct graph transformations over models represented in relational databases. The most important difference to prior work [VFV05] is the application of incrementality to improve the performance of the graph pattern matching phase (see Incrementality in Section 4.3.1); while [VV04] is extended by (i) the detailed description of a universal procedure (inspired by TREAT [ML91]) that achieves incrementality for any GT program (see Declarativity in Section 4.3.1), (ii) the non-interference with existing programs that manipulate the graph model (see Compatibility in Section 4.3.1), and (iii) some pattern language features (see Section 4.3.4).

Incremental pattern matching requires (a) caches preserving previously computed results and (b) mechanisms to update such caches upon changes of the input. The first is achieved by using additional database tables to store cached relations. One possible solution to the second problem could be a global policy stating that all operations writing to the graph model must conclude by explicitly invoking cache maintenance routines that propagate the changes to the pattern match results [VV04]. However, in order to satisfy the goal of Compatibility (see Section 4.3.1), the presented novel solution does not require the modification of any existing programs manipulating the graph model. This approach
4.3. INCREMENTALITY ON TOP OF EXISTING RELATIONAL DATABASES

Figure 4.3: Overview of the mapping process

employs database triggers instead to refresh the contents of the cache tables in an event-driven fashion, after arbitrary transactions manipulating the model.

An algorithm is provided to generate SQL code from the graph transformation program, in accordance with the Declarativity goal (see Section 4.3.1). The proposed approach, depicted in Figure 4.3, has three main phases:

1. **Mapping between the graph metamodel and the relational schema** is the well-known problem of Object-Relational Mapping (ORM) [GMUW08] executed in either direction. To retain Compatibility (see Section 4.3.1) with systems already deployed in RDB, a relational schema can be used as input if available. In a particular ORM strategy for example, each class \( C \in \text{Cls}_{\text{Str}} \) in the structural metamodel corresponds to a RDB table \( \text{Entity}_C(\text{ID}, \ldots) \), with a separate column for each associated data attribute name \( \text{attr} \in \text{Fea}_{\text{Val}} \), and a primary key column as unique identifier. Structural entities appear as a row in the table of their class and those of the superclasses, logically connected by the common identifier. An association \( R \in \text{Fea}_{\text{Str}} \) from class \( \text{SrcT} \) to type \( \text{TrgT} \) corresponds to a separate table \( \text{Relation}_R(\text{ID}, \text{SrcID}, \text{TrgID}) \) (or, alternatively, \( \text{Relation}_R(\text{SrcID}, \text{TrgID}) \) if parallel associations of the same type are not permitted, and relations are identifiable by source and target). Each row of \( \text{Relation}_R \) represents one structural relation instance, and the two foreign key columns reference the identifiers of the source and target entity (rows in \( \text{Entity}_{\text{SrcT}} \) and \( \text{Entity}_{\text{TrgT}} \), respectively). A feature \( R \) with a multiplicity of 1 could alternatively be associated with a column of \( \text{Entity}_{\text{SrcT}} \) referencing the \( \text{TrgT.ID} \) key of the single target entity. Note that there are several other possible ORM methods, and the general approach outlined in the following is applicable to all of them.

2. **Cache tables and maintenance triggers for patterns** are the main contribution for this thesis. A database table \( \text{Memo}_P \) is created for each pattern \( P \) to preserve its match set. Unlike [VV04], incremental maintenance of the contents of these tables is performed by database triggers, that are automatically generated from the declarative pattern description. The triggers are activated by updates to the database tables representing the graph elements, and potentially the other cache tables as well. The solution is described in detail in Section 4.3.4.

3. **Mapping GT rules to stored procedures** is performed according to [VFV05], no modifications to the existing approach are required. The main idea is that the application of the GT rule
4.3.3 Basic pattern matching over a relational database

In the following, a basic solution is described for matching graph patterns on graph models stored in relational databases; this procedure will be the building block of the incremental solution in Section 4.3.4. The original idea can be found in [VFV05] along with more details and examples.

Let us assume that the metamodel has already been mapped into a relational schema (see Section 4.3.2). For a simple graph pattern $P$ consisting of entity and relation constraints, the SQL view definition $View_P$ is a relational join operation on several tables that yields the matches of pattern $P$. This solution from [VFV05] is not yet incremental, as each evaluation of $View_P$ will re-execute the join.

With the ORM mapping in Section 4.3.2, for each class constraint $\langle v, C \rangle \in \mathcal{C}_{str}^\text{ent}$ that restricts the pattern variable $v$ to class $C$, $View_P$ will involve the table $Entity_C$ as a participant in the join operation. Likewise for each association constraint $\langle u, r, v, R \rangle \in \mathcal{C}_{str}^\text{rel}$ expressing that there is a relation $r$ of type $R$ from pattern variable $u$ to $v$, the join in $View_P$ will include the table $Relation_R$. The incidence of entities and relations are enforced by join conditions. A single table can appear several times in the join, if the graph pattern has multiple association or class constraints with the same type. Equality and inequality constraints are easily checked by comparing identifiers.

Example 29 The pattern inhibits of Figure 3.2 is mapped into the SQL view definition $View_{\text{inhibits}}$ in Listing 4.1. The pattern expresses two relation constraints; therefore the join expression involves each of $Relation_Marking$ and $Relation_{\text{InhibitorArc}}$ exactly once. Aliases $TM$ and $TH$ are assigned to the corresponding table rows constituting a result row, partly for brevity, partly to avoid name clashes if there are multiple relation constraints of the same type. There are no entity constraints to restrict the types of variables to a subtype of the feature owner/range, therefore no tables such as $Entity_{\text{Place}}$ are involved.

4.3.4 Incrementality using cache tables and triggers

In the following, an approach is introduced for incrementally maintaining match sets of graph patterns in relational databases. The presented solution adapts the well-known TREAT algorithm [ML91] to relational databases and a basic graph pattern formalism. The reasoning behind the choice of this algorithm over Rete is that it is more suited to the RDB environment, since the RDB provides an efficient, optimized SQL query infrastructure that can fulfill the role of TREAT maintenance, alleviating the need for Rete query planning.

Listing 4.1: SQL View Definition to return matches of the pattern inhibits (subscripts indicated by underscores)

```sql
CREATE VIEW View_inhibits AS
SELECT TM.SrcID AS P, TM.ID AS M, TM.TrgID AS K, TH.ID AS H, TH.TrgID AS T
FROM Relation_Marking AS TM, Relation_InhibitorArc AS TH
WHERE TM.SrcID=TH.SrcID
```

is decomposed into individual elementary model manipulation operations (see Section 2.1.5), which are then simply transcribed into SQL manipulation commands. The resulting sequence is then automatically assembled into an SQL stored procedure that takes the LHS match as input.
For each pattern $P$, a table $Memo_P$ will be created that caches the matches of $P$. These tables will store exactly the match sets of the associated patterns, thus the algorithm has minimal cache (this is a property of the TREAT algorithm). The previously defined view $View_P$ can be used to initialize the table (or alternatively, it can be constructed from an empty table by the incremental maintenance mechanisms). For each class $C$ referenced by $P$, the match set of the pattern may change whenever entities of type $C$ are created or deleted (or retyped). The match set may also change when relations of type $R$ are created or deleted, provided $R$ is mentioned in $P$. All of these changes are observable as row insertions / deletions in tables $Entity_C$ or $Relation_R$, the entirety of which consists the graph delta. Therefore database triggers can deal with maintaining the match set. Triggers for row insertion and deletion are registered for each entity table $Entity_C$ or relation table $Relation_R$ that $P$ depends on. Taking the graph delta that has triggered them, they execute the maintenance routine to compute $\Delta MatchSet^P$, and update $Memo_P$ accordingly. More formally:

**Definition 72 (TREAT-based stateful pattern matcher in relational database)** The TREAT algorithm over a relational database is a stateful pattern matcher $PM_{TREAT-RDB} = \langle STREAT-RDB, s_{TREAT-RDB}^0, Match_{TREAT-RDB}, Maintain_{TREAT-RDB} \rangle$ where

- the state $s$ consist of a table $Memo_P$ for each pattern $P$ that needs to be fully cached (and, where needed, additional index structures for efficient retrieval);
- $s_{TREAT-RDB}^0$ contains empty tables $Memo_P$;
- the match routine $Match_{TREAT-RDB}(P)$ simply queries the cache table $Memo_P$;
- the TREAT maintenance routine $Maintain_{TREAT-RDB}(\delta)$, executed by triggers registered in the RDB, incrementally updates each affected cache table $Memo_P$ using insertions and deletions.

The following paragraphs elaborate the maintenance routine of the TREAT approach.

**Relation insertion.** First, let us consider the case where there is an association constraint $c = \langle u, r, v, R \rangle \in C_{rel}^R$ and the change is the creation of a new relation instance of type $R$, appearing as a new row $\langle ID_{Src}, ID_{Trg} \rangle$ in table $Relation_R$. The delta will be the set of matches that contain the newly created relation. Therefore the trigger will insert the result of query $\Delta^+_c$ into $Memo_P$, where $\Delta^+_c$ is a modified ("seeded") version of $View_P$, restricted to the new matches that are produced by this relation insertion (using seeded pattern matching). $\Delta^+_c$ is formed by omitting the $Relation_R$ operand that corresponded to the association constraint $c$ from the join expression computing $View_P$, and substituting its source and target identifier values respectively with the triggering $ID_{Src}$ and $ID_{Trg}$. These input values reduce the cardinality of the result relation significantly, making incremental maintenance efficient.

If the pattern contains $k$ relation constraints for the type $R$, then $View_P$ is seeded similarly for each of them, and the delta is the union of the results. However, one must take caution with the matches where several variables take the same value. In this case it is also possible that the new relation produces a match by simultaneously satisfying several of these $k$ constraints. One possible solution is to compute the delta as the union of $2^k - 1$ branches, depending on which of the variables are substituted with the given new relation (at least one of the $k$ is the new relation). There are other solutions that are not detailed here for brevity.
CHAPTER 4. ADVANCED INCREMENTAL PATTERN MATCHING

SELECT ID_Src AS P, ID_new AS M, ID_Trg AS K, TH.ID AS H, TH.TrgID AS T
FROM Relation_InhibitorArc AS TH
WHERE ID_Src=TH.SrcID

Listing 4.2: Seeded SQL query for computing the delta (relation insertion, case of Marking M)

Relation deletion. With deletion, there are two basic options in variants of TREAT. The straightforward solution is to implement deletion triggers that are symmetric to creation triggers, evaluate seeded $\Delta_-$ queries, and remove the results from $Memop$. A potentially faster solution would be to directly scan $Memop$ and remove all matches that were produced by the deleted relation. This approach may speed up deletion, but it will not improve the overall update overhead by more than 50% as insertion triggers are unaffected. Furthermore, selecting affected rows from $Memop$ is only fast if the appropriate index structures are established on $Memop$, which will in turn exert a maintenance overhead (though some of these indexes would be necessary for handling pattern composition, see Section 4.3.5).

Entity manipulation. Insertion and removal of instances of classes are processed analogously to relation operations. In addition to the creation and deletion of entities, which is only possible when there are no incident relations, this might include entity retyping as well. However, retyping is often disallowed, in which case these trigger will only run if there are no relations to join the entity against. Thus it is possible to improve performance of entity triggers by only seeding patterns where the corresponding entity constraint is isolated (without relation constraints incident on the same variable); if there are no such patterns, no trigger is required at all. Given the nature of the graph pattern formalism, isolated entity variables in the LHS are uncommon, save for the case where the entire LHS is a single entity (which makes the trigger trivial anyway).

Example 30 Let us consider pattern inhibits again (see Figure 3.2), for which View inhibits is constructed in Example 29. In order to maintain the cache table $Memo_{inhibits}$, triggers will have to be registered for the insertion (and, symmetrically, deletion) of rows into tables Relation Marking and Relation InhibitorArc. Since both these tables were involved in the original join expression a single time, each row insertion / deletion involving them has to be considered in a single way only to contribute to a new match / invalidated match. Focusing now on the Marking relation constraint $c_M$, in case of the insertion of the row $\langle ID_{new}, ID_{Src}, ID_{Trg} \rangle$ into Relation Marking, the trigger will evaluate a seeded query to obtain the delta relation $\Delta_{+}^{c_M}$, and add the contents of the delta to the cached match set $Memo_{inhibits}$. The contents of the seeded query are displayed in Listing 4.2; it simply finds the InhibitorArc instances connected to the Place where the new token was put.

4.3.5 Advanced pattern language features

Attribute checks. Attributes are columns of the EntityC tables, and value assignment constraints are translated into accessing the appropriate attribute of the table. Polymorphism (type inheritance) may require the query to first join the table EntityC to the table EntityC' if C' is the supertype of C that defines the particular attribute. Data predicate constraints are enforced as attribute checks in the WHERE SQL clause.

Composition. Pattern composition is not supported in [VV04]. The proposed approach treats a pattern composition constraints similarly to association constraints. If such a constraint in pattern Caller references pattern Called, the Memo Called table will participate in the join operation
4.3. INCREMENTALITY ON TOP OF EXISTING RELATIONAL DATABASES

computing the deltas of $\text{MemoCaller}$. As columns in $\text{MemoCalled}$ will be used as join keys, SQL data definition commands have to be issued that build index structures on these columns to improve performance. Triggers on $\text{MemoCalled}$ will also be registered to propagate the changes between the match sets of the patterns. Recursive graph patterns cannot be handled this way, just like they cannot be handled by Rete (see Section 2.2.2.3); and also because many RDB environments disallow circularity in triggers. On the other hand, using pattern composition with INC can in some cases have a very beneficial effect on performance, as the computed result of the called pattern can be reused many times during the maintenance of the calling pattern.

**NAC support.** In the presence of a NAC, the join operation computing the match set will involve an outer join of $\text{MemoNAC}$ which checks that there are no corresponding matches of the NAC pattern (thereby implementing an anti-join). Insertion triggers on $\text{MemoNAC}$ will delete matches of the positive pattern; and deletion triggers will produce new matches if no remaining rows of $\text{MemoNAC}$ inhibit it.

**Parameters and disjunction.** Disjunctive patterns and non-parameter variables are not handled in [VV04]. There are a number of ways to implement these in SQL. One solution is to create a separate internal cache table for each of the pattern bodies, which will be updated by triggers according to the procedures described above. An externally visible cache table will also be created for the pattern itself, that contains the union of the individual match sets (projected onto the set of parameter variables). This table will be incrementally maintained by triggers defined on the internal cache tables.

4.3.6 Performance observations

The paper [7] reports on performance measurements carried out on graph transformation sequences. The benchmarks included a model-to-model synchronization task, a model simulation task, and finally a synthetic example where incremental approaches are known to be at a disadvantage. The performance of a prototype implementation of this approach was measured against non-incremental pattern matching using SQL queries, as well as incremental (Rete-based) and non-incremental solutions in ViATRA2.

The measurements confirmed that the RDB model representation is generally slower than VPM but more compact in memory. Furthermore, the TREAT approach used for the RDB-based incremental matcher also makes a similar trade when compared against Rete: it uses less cache memory, though it may have worse performance. Finally, in the two use case-oriented benchmarks (but not the synthetic counterexample), incremental approaches were demonstratively faster than their non-incremental alternatives, at the cost of cache memory.

The Petri net firing benchmark of Section 3.5.1 on page 67 and [16] was essentially reused as the model simulation case; see Figure 4.4 for a performance comparison of the four approaches on the same hardware. For more discussion and the other two case studies, see paper [7].
Figure 4.4: Petri net firing benchmark, average from 1000 transition firings
Chapter 5

Incremental Model Queries over Industrial EMF Models

As a leading industrial modeling ecosystem, EMF provides automated code generation and tooling (e.g. notification, editor) for model representation in Java. It is successfully applied in a wide range of industries, including automotive, aerospace, finance, energy, and health. This provides plenty of motivation to bring academic modeling results into industrial practice. While earlier chapters have presented graph patterns as a formal query representation, along with Rete as a theoretical algorithm, here they will be adapted to the industrial platform of EMF in form of the model query tool EMF-IncQuery. The chapter will present an EMF-specific query language syntax, an translation mechanism that integrates the Rete-based approach to technological constraints of EMF, and experiments that measure performance in an industrial context.

This chapter partially follows [10] and [11].

5.1 Platform and case study

Here, I give a quick overview of the relevant technical aspects of the EMF modeling platform, and then proceed to introduce an example problem that will be used in Section 5.2 to motivate the construction of model queries over EMF.

5.1.1 EMF technical preliminaries

An introduction to EMF model elements, metamodels and idiosyncrasies of the EMF modeling paradigm was given in Section 2.1.6. Some further technical details are presented below.

EMF model elements are organized into Resources, which can be loaded from and saved to some location and format (such as an XMI file, a file of some domain-specific format, or a registered metamodel from a plug-in component) that is identified by a URI (unique resource identifier). A Resource holds a containment tree (or sometimes containment forest) of EObjects and is closed with respect to containment in both directions. However, there can be non-containment references (cross-references) crossing the boundaries of Resources. A group of Resources that are also closed (with some minor exceptions) with respect to cross-references is a ResourceSet. For the sake of conciseness, Resources are typically considered to be contained within their ResourceSet, and top-level (containerless) EObjects are considered to be contained within their respective Resources in the same way that other EObjects
are contained (transitively) in them, so that the entire ResourceSet forms a single unified containment tree. This containment hierarchy is illustrated in Figure 5.1.

![Figure 5.1: Illustration of the EMF containment hierarchy](image)

EObjects are analogous to structural entities in the terminology of Section 2.1. It is important to keep in mind that structural relations (references in EMF terminology) are not individual objects in EMF, and therefore one cannot refer to a specific relation instance, except by the triple $(source, association, target)$. The same holds for value assignment relations (EObject fields) as well.

Only a restricted set of elementary model query operations introduced in Section 2.1.5 is supported (efficiently) in EMF, the rest can only be evaluated via exhaustive search. In particular, $Query(E : C)$ and $Query(E :: C)$ can only be evaluated if $E$ is specified as an actual EObject (i.e. it cannot be the wildcard $\ast$). $Query(E_s \xrightarrow{(R:F)} E_t)$ can only be evaluated if $E_s$ and $F$ are both given as a concrete value, and $Query(E_s \xrightarrow{(R::F)} E_t)$ is equivalent to it as there is no feature subtyping. However, assuming an imaginary association contains that is thought of as the common supertype of all containment associations (ERefereces), $Query(E_s \xrightarrow{(R:contains)} E_t)$ can be evaluated if either $E_s$ or $E_t$ is given as a concrete EObject, i.e. both the container and contained objects of an EObject can be found out.

EObjects support notification adapters. If such an adapter is registered at an EObject, it will receive notifications whenever an outgoing relation is added to / removed from that entity. The notification message indicates whether the relation was added or removed (or retargeted), the source EObject, the type of the relation (EReference or EAttribute) and the old/new target entity (which is a data value in case of EAttributes). Similarly to EObjects, ResourceSets and Resources are also Notifiers; one can attach notification adapters and receive notifications when the set of contained Resources, respectively top-level EObjects is changed.
5.1. PLATFORM AND CASE STUDY

5.1.2 Motivating example: security requirements

5.1.2.1 Problem Domain

The presented motivating scenario is from the domain of security requirements engineering (see Section 1.1.4.1), inspired by the Air Traffic Management case study of the SecureChange European research project [EU 12]. A requirements model assists security engineers to capture security related aspects of the system, to analyze the security needs of the stakeholders, and to argue about potential security threats. The concepts of a security requirements modeling language such as SecureTropos [M02] typically include actors (stakeholders and their human and machine agents), resources (e.g. security-critical information assets) provided by actors, goals (functional, security, etc. requirements) wanted by actors, and tasks performed by actors. Relationships include tasks to fulfilling goals; trust relationships between actors; and delegation of responsibility over resources, goals or tasks.

Example 31 See Figure 5.2 for a simplified extract of the SecureChange requirements metamodel [MMP+11] that will be used for this case study. The metamodel has been formulated in Ecore, and displayed here using the Ecore Diagram visual concrete syntax, which was designed to resemble UML class diagrams. Of the main elements of Ecore (see Section 2.1.6), an EClass is graphically depicted as a box (with the name shown in the top compartment), an EReference is depicted as an edge (labeled by name and multiplicity) between the boxes associated with the owner and range types, and each EAttribute is listed within the middle compartment of the box representing the owner EClass. Subtyping is depicted by an arrow ending in a hollow triangle arrowhead, pointing from the subclass to the superclass; only a transitively reduced subset of supertype relationships needs to be explicitly shown, so that they are sufficient to imply the entire supertyping partial order.

An important role of security requirement models is to support reasoning on security properties in an early phase of system development. To formalize static security constraints, graph patterns are used as a query language.

The motivation behind providing incremental model queries over EMF is that security requirement tooling should provide efficient, incremental constraint evaluation and feedback for the engineers even in the early stages of requirements modeling. As a benefit of incrementality, requirement models can be validated continuously, i.e. security violations are indicated on-the-fly during requirement engineering, retaining quick response times even for complex queries and large requirements.
models. Such a service provides a shortened feedback cycle, raising the efficiency of the engineering process.

**Example 32** Figure 5.3 shows an example security requirement model with two actors $A_1$ and $A_2$; three goals $G_1$, $G_2$ and $G_3$; four tasks $T_1$, $T_2$, $T_3$, $T_4$; and the wants, does, fulfills edges between them, as well as mutual trust between the two actors. As the abstract model of [MMP+11] has no associated visual concrete syntax, symbols were loosely based on Si*/Tropos [MMZ07] diagrams: actors are depicted here by large red discs, goals by rounded green rectangles, and tasks by blue hexagons. Each goal also has a redundancy requirement enclosed by curly braces.

![Figure 5.3: Example security requirement model with redundancy requirements](image)

Another example tied to a real case study will be shown in Section 6.1.3.3, and more elaborate models from a related case study will be shown in Section 6.1.3.

### 5.1.2.2 Analysis Tasks

Early-stage analysis of requirements models is carried out by (local and global) model queries. Support can range from finding violations of structural semantic constraints that represent security properties of the model, to generating reports that guide the engineer to fix these problems.

One challenge where early-stage analysis is beneficial is to detect violations of the trusted path security constraint. The context is the following: a valuable data asset is provided by one actor, and is eventually delivered to a recipient actor, through potentially unreliable intermediate actors. A security goal requires the protection of the integrity and confidentiality of this data resource. The trusted path security constraint states that either a trusted actor has to perform an action that explicitly fulfills the goal (e.g. time-stamping, digital signature and encryption), or else the entire data path must be trusted; indirect trust is permitted. The challenge is to formulate a query `noTrustedPath` which finds the violations of this security constraint, so that the security problem can be reported to the engineer.

A second application of model queries is related to the redundancy security constraint. Redundancy is important for resilience against failures and attacks, and is therefore an integral part of security; thus requirements often have a minimal degree of redundancy associated with them. For example, the availability requirement of a service task or data asset can be augmented with the demand of triple modular redundancy, i.e. 3 replicas of the given data / service must be available. A goal with the redundancyRequirement attribute set must be fulfilled by at least this number of separate tasks (performed by trusted actors). Two queries will be formulated in connection with this security constraint: (a) `redundancyViolated` to find goals whose redundancy requirement is not met, and (b) `totalMissingReplicas` to compute an actor-centric progress indicator that informs of the total number of missing replicas for all the goals wanted by a given actor.
5.2. EMF MODEL QUERIES BASED ON GRAPH PATTERNS

All three of these constraints will be formulated in the sequel using a query language.

**Example 33** Given the example model depicted in Figure 5.3, the first redundancy query `redundancyViolated` will find goals $G_1$ and $G_2$. The second redundancy query `totalMissingReplicas` will return 3 for actor $A_1$ (as two fulfilling actions are missing for $G_1$ and one for $G_2$), and 0 for actor $A_2$.

5.2 EMF model queries based on graph patterns

EMF-InCQuery [11,9] is a framework with a language for defining declarative local and global queries over EMF models, and a runtime engine for executing them efficiently without manual coding. The query language of EMF-InCQuery is built on the concepts of graph patterns and partially reuses the syntax of ViTrA2 [VB07] as a concise and easy way to specify complex structural model queries.

In the subsequent sections, I will present the syntax of EMF-InCQuery step-by-step, by presenting a solution to the motivating problem. The new language introduces some significant semantic extensions over its precursor ViTrA2, as well as syntactic sugar for conciseness. The two main areas where EMF-InCQuery differs from the original VTCL syntax are the structural/navigational language elements and the handling of attributes and arithmetic expressions.

5.2.1 Structural constraints

The graph pattern based query language of EMF-InCQuery is based on the graph pattern formalism of Section 2.2. The type system is based on Ecore: pattern constraints use EClasses as classifiers, EReferences and EAttributes as (association resp. attribute name) features. Pattern variables will be mapped to EObjects of the instance model (or values from the data algebra, see later). All patterns conform to the notion of disjunctive pattern (see Section 2.2.2), even if they have a single pattern body; thus not all variables have to be exposed as parameters.

Due to the fact that EMF lacks relation objects, the most important deviation from the formalization in Section 2.2 is that relation constraints do not have a relation variable. Or as an equivalent alternative, the relation variable exists but must not be used anywhere else, i.e. it is not allowed as a parameter and cannot be in the domain of any other constraint. In either case, the textual syntax for relation constraints simply omits the relation variable.

**Example 34** The first example demonstrates the structural pattern constraints of the EMF-InCQuery language. The trusted path security constraint is checked by the graph pattern `noTrustedPath` defined in Listing 5.1. As introduced in Section 5.1.2.2, the trusted path security constraint states that the delegation paths of resources protected by security goals must entirely consist of trusted actors, unless there is an explicit fulfilling task carried out by a trusted party. Line 1 introduces the name of the pattern and lists the four parameter variables `concernedActor`, `secGoal`, `asset` and `untrustedActor`; all other variables will be local to a body. In the following lines, curly braces enclose the single pattern body; multiple bodies would be separated by the keyword `or`. The body contains pattern constraints separated by semicolons:

- Line 3 represents an entity constraint that states that variable `secGoal` corresponds to an EObject of type `SecurityGoal`.
- Line 2 expresses a relation constraint that navigates from variable `concernedActor` along an EReference of type `wants`, and the EObject reached that way should be the one associated with variable `secGoal`. Note that there is no separate variable for the relation itself.
CHAPTER 5. INCREMENTAL MODEL QUERIES OVER INDUSTRIAL EMF MODELS

Listing 5.1: Violations of the trusted path security constraint, EMF-specific syntax

```
pattern noTrustedPath(concernedActor, secGoal, asset, untrustedActor)={
    Actor.wants(concernedActor, secGoal);
    SecurityGoal(secGoal);
    SecurityGoal.protects(secGoal, asset);
    Actor.provides(providerActor, asset);
    find transitiveDelegation(providerActor, untrustedActor, asset);
    neg find trustedFulfillment(concernedActor, _anyActor, _anyTask, secGoal);
    neg Actor.trust*(concernedActor, untrustedActor);
}
```

- Similarly, Line 4 and Line 5 introduce additional structural relation constraints. A chain of two or more such relation constraints can be abbreviated as e.g. `Actor.wants.protects(concernedActor, asset);` instead of Line 2 and Line 4; the more verbose form was chosen here so that `secGoal` can be a parameter variable.

- Line 6 shows an example for pattern composition. The called pattern is `transitiveDelegation` (defined elsewhere) and the variables `providerActor`, `untrustedActor`, and `asset` are mapped to its parameters in the composition mapping (see Section 2.2.2.3).

- Line 8 expresses a NAC in a negative pattern call constraint; it can alternatively be understood as the equivalent case of a negative subpattern holding a single pattern composition constraint that call pattern `trustedFulfillment` (defined elsewhere). Variables `_anyActor` and `_anyTask`, as indicated by a Prolog-style underscore, are single-use variables appearing only in the negative subpattern. Altogether pattern `noTrustedPath` matches for a given `concernedActor` and `secGoal` only if there exists no substitution of `_anyActor` and `_anyTask` that satisfies the call of `trustedFulfillment`.

- Line 7 computes a transitive closure inside a negative subpattern. The closure is indicated using the transitive closure operator (the symbol is `*` for reflexive transitive closure, while `+` would denote irreflexive closure). Here, the reflexive operator is applied to the EReference `Actor.trust`, instead of a pattern call of a binary pattern (which would be only possible in case of irreflexive transitive closure).

Two major limitations of the core EMF API are the lack of (i) efficient enumeration of all instances of a class regardless of location, and (ii) backwards navigation along uni-directional references. As seen here, the structural graph constraints of EMF-InCQuery can provide these missing features for end-users, via EMF-InCQuery Base [SU].

5.2.2 Attribute and arithmetic constraints

Example 35 The next example demonstrates the value assignment and data predicate pattern constraints of the EMF-InCQuery language. The redundancy security constraint is checked by the graph pattern `redundancyViolated` defined in Listing 5.2. As introduced in Section 5.1.2.2, the redundancy security constraint states that a goal `goal` with a `redundancyRequirement` attribute must be fulfilled at least as many times (by trusted actors) as specified by the attribute value. First we investigate the helper pattern `redundantReplicas` that counts the number of replicas fulfilling the given goal wanted by the given actor, and also extract the required redundancy. Line 6 asserts a
5.2. EMF MODEL QUERIES BASED ON GRAPH PATTERNS

Listing 5.2: Violations of the redundancy security constraint

```
pattern redundancyViolated(concernedActor, goal)={
  find redundantReplicas(concernedActor, goal, redundancyFound, requiredRedundancy);
  check (redundancyFound < requiredRedundancy);
}

pattern redundantReplicas(concernedActor, goal, redundancyFound, requiredRedundancy)={
  Actor.wants(concernedActor, goal);
  Goal.redundancyRequirement(goal, requiredRedundancy);
  redundancyFound == count find trustedFulfillment(concernedActor, _anyFulfillerActor, _anyTask, goal);
}
```

structural relation (reference) of association type `Actor.wants`, while line 7 asserts a value assignment of type `Goal.redundancyRequirement`. Both kind of relation constraints are expressed by similar syntax, but the type is an EReference in the first case and an EAttribute in the second. As a consequence, `requiredRedundancy` will be a data entity variable of type EInt (equivalent to Integer). Lines 8-9 express an aggregate pattern constraint (see Section 2.2.2.4) that aggregates the count of those matches of pattern `trustedFulfillment` that are incident on the given actor and goal, and the aggregate result is stored in `redundancyFound`, which is once again a data entity variable of type EInt. Note the use of the underscore notation (similar to the NAC in Listing 5.1) for variables `_anyFulfillerActor` and `_anyTask` that are not part of the main pattern `redundantReplicas`, merely aggregated over. This pattern `redundantReplicas` is called by pattern `redundancyViolated` on Line 2. Line 3 contains the attribute check verifying that the value of variable `redundancyFound` is less than the value of `requiredRedundancy`; this is actually a relation constraint between the two variables, where the type is the data predicate `<` (more complicated checks would be interpreted as data predicates with higher arity).

The pattern `redundancyViolated` described in Listing 5.2 identifies violations of this security constraint, using a helper pattern.

Example 36 The secondary challenge is to provide an actor-centric indicator report on the number of missing replicas; the solution is shown in Listing 5.3. The number of further replicas needed that are trusted by a given actor to fulfill a given goal is computed by pattern `missingReplicas`. Line 6 reuses pattern `redundantReplicas` defined by Listing 5.2. Line 7 computes the difference between the required and present degrees of redundancy, and stores the result in variable `missing`; this line expresses a data predicate constraint of ternary arity that is functionally determined (see Definition 18) by two of its variables. The check in line 8 uses variable `missing` and the value literal 0; as it introduces no new variable, it does not have to be functionally determined (hence the use of the `check()` syntax instead of `eval()`). Pattern `totalMissingReplicas` uses the aggregate expression in line 3 to add up all these missing replica counts through all goals of a given actor; `sum(missing)` specifies that it is variable `missing` that should be summed (i.e. the match is mapped into the Abelian group of integers as the value of `missing`). Note the use of the underscore notation for variable `_anyGoal` that is not part of the main pattern `totalMissingReplicas`, merely aggregated over.
102  

CHAPTER 5. INCREMENTAL MODEL QUERIES OVER INDUSTRIAL EMF MODELS

Listing 5.3: Missing replicas per actor and goal

```plaintext
pattern totalMissingReplicas(concernedActor, totalMissing) = {
  Actor(concernedActor);
  totalMissing == sum(missing) find missingReplicas(concernedActor, _anyGoal, missing);
}

pattern missingReplicas(concernedActor, goal, missing) = {
  find redundantReplicas(concernedActor, goal, redundancyFound, requiredRedundancy);
  missing == eval(requiredRedundancy - redundancyFound);
  check (missing > 0);
}
```

5.2.3 Query language structure

Summarizing the preceding example-driven introduction, the abstract structure of the query language is presented here, without going into the details of the actual grammar.

Patterns have a name, a list of named parameter variables (with optional type constraints), and one or more pattern bodies. A pattern body is a list of pattern constraints that are expressed over arguments.

Certain constraints are enumerable constraints; this means that the fulfilling value combinations can be enumerated from the model, without any prior knowledge. These include the entity constraint, the relation constraint, longer path expressions (consisting of a chain of two or more relation types), the pattern call and transitive closure (which is the irreflexive closure of a binary enumerable constraint). Non-enumerable constraint types include NAC (which is the negation of an enumerable constraint), equality, inequality and extensible Boolean-valued check expressions on variables.

The argument expressions can be variables (which are either local to the body or one of the pattern parameters), constant literal values and computed values. Computed values include extensible expression evaluation on variables, and aggregation of an enumerable constraint by a suitable aggregator.

All of these language elements have been demonstrated by the above examples.

5.3 Integrating incremental pattern matching to EMF

Here an approach is described for realizing the Rete-based incremental matcher above EMF, facing the technical constraints described in Section 5.1.1, in order to match patterns defined in the language of Section 5.2. According to Chapter 3, sustaining a stateful pattern matcher requires the modeling platform to provide elementary model query operations, and to invoke the maintenance routine with graph deltas upon each change to the model. The following paragraphs investigate how this can be achieved over EMF.

5.3.1 EMF as graph model with elementary queries

The first issue to address is that of elementary model queries. Since EMF does not provide an efficient implementation for all elementary query cases, a straightforward solution is to extend the platform by establishing index structures that can efficiently answer these queries. For example, one such index may map each EClass (or, for better performance, a restricted subset of EClasses) to the set of its instance EObjects. Another one may index EObjects of a given type according to the value of one of their EAttributes, or keep track of the references (structural relations) incoming to an EObject. These
index structures can be initialized by a traversal of the EObjects constituting the model, and they can be incrementally maintained based on the graph delta by the maintenance routine of the stateful matcher (before updating the Rete structure).

The expanded set of efficient elementary model queries is publicly available as a separate service called EMF-IncQuery Base [9][SU], since it is useful in a wide range of applications beyond the initialization of Rete input nodes. The following paragraphs will discuss how EMF models can be traversed to initialize these indexes (which in turn will be used to initialize the input nodes of Rete), and how to obtain a graph delta upon changes to the EMF model so that the maintenance of both Rete and these indexes can be carried out.

While there can be many EObject instances existing simultaneously within the system memory, they probably should not all belong to the same model; and even if they do, there should be a way of finding and enumerating them, so that the model can be traversed and EMF-IncQuery Base caches can be initialized. Therefore, a query engine must have a way to find out which model elements constitute a model, i.e. which elements are within the scope of the pattern matcher. The most practical choice is to define an EMF instance model as the contents of the containment forest below a given set of roots (which may be EObjects, Resources and ResourceSets). The typically case is the containment tree of a single ResourceSet. There may be cross-references egressing or ingressing the boundary of this containment forest; the question whether they should be considered part of the model will not be discussed here, but (normally) there are no such references if the model is a single ResourceSet.

Example 37 Suppose that the requirements model described in Example 32 on page 98 resides in an EMF Resource $R_1$ within ResourceSet $S$. Suppose also that there is a separate Security Goal $G_4$ with redundancy requirement 2 that resides in EMF Resource $R_2$, which is not contained in the ResourceSet $S$. If this ResourceSet $S$ is the scope of EMF-based graph model $g$, then for instance $g.Query(∗ :: SecurityGoal).out = \{ ⟨G_1, SecurityGoal⟩, ⟨G_2, SecurityGoal⟩, ⟨G_3, SecurityGoal⟩ \}$, since the fourth goal instance is outside the scope.

5.3.2 Translating from EMF notifications to graph delta

Containment hierarchies from given starting points can be traversed using the allowed elementary model query operations, therefore the contents of an EMF model can be discovered. Determining the graph delta is more difficult.

An EMF model can be changed by modifying members (EAttribute values, EReference targets) of an EObject, or by adding/removing EObjects to/from a Resource, adding/removing Resources to/from a ResourceSet (some miscellaneous cases omitted). Upon the change, an EMF change notification message emanates from the model. If a relation is inserted or removed, the notification contains the direction of change (insertion/deletion) and the source, target and type of the changed relation (only the relation identifier itself is omitted, which does not exist in EMF). If the contents of a Resource or ResourceSet is changed, the notification message is similar, but lacks a relation type. As described in Section 5.1.1, the change notification is then delivered to notification listeners/adapters registered at the source Notifier. Then the challenge is to translate from this notification schema to graph deltas as required by the incremental pattern matcher.

The first approximate solution is to attach notification listeners to each EObject in the model scope, and then treat the received notifications as graph model deltas. This initial solution has some shortcomings:

- Deleting a containment relation would remove the whole containment subtree of the target object from the model (according to the above definition of model boundaries); the model delta
should therefore contain several entity and relation deletions even though only a single notification is received about the deletion of the containment relation.

• Conversely, a whole containment subtree can be inserted into the model at once with a single operation.

• If new entities are added to or removed from the models, the set of Notifiers that have registered adapters must be adjusted accordingly, so that no notifications are missed from newly inserted Notifiers, and no irrelevant/misleading notifications are received from Notifiers no longer belonging to the model.

The chosen solution relies on EContentAdapter, an advanced notification adapter mechanism shipped with EMF. An EContentAdapter, when attached to a Notifier, will also attach itself recursively to all contained Notifiers; conversely, when it is removed, it will also remove itself from all contained elements. Furthermore, while attached to a Notifier, it will also monitor notifications of containment relations to maintain itself as an adapter attached to all contained elements. Consequently, after an EContentAdapter is manually attached to one or more Notifiers, it will keep itself attached exactly to those Notifiers that constitute the containment forest rooted at the given elements, i.e. the extent of graph model. This mechanism can be extended by the following additional notification handling behavior to meet our needs:

• Relation notifications will be treated as relation insertions/deletions in the graph delta.

• When the adapter is attached to a new EObject, the entity itself as well as all outgoing relations are considered as insertions in the graph delta.

• When the adapter is removed from a new EObject, the entity itself as well as all outgoing relations are considered as deletions in the graph delta.

There are still some minor technical details that require attention in an implementation, but are omitted here for brevity. These include EMF derived features in general and special cases such as FeatureMaps; exceptional situations when a ResourceSet is not closed with respect to cross-references; and so on.

Example 38 Continuing from Example 37, the EContentAdapter is attached to ResourceSet $S$, Resource $R_1$, and the EObjects contained in $R_1$ (namely $A_1$, $A_2$, $G_1$, $G_2$, $G_3$, $T_1$, $T_2$, $T_3$, $T_4$); it is not attached to $R_2$, since it is not contained in $S$.

Supposing now that Resource $R_2$ is loaded within ResourceSet $S$, a single EMF change notification is received at $S$: an insertion, with $S$ as source, $R_2$ as target, and no feature (since this is a change of the ResourceSet contents). As a consequence, the EContentAdapter installs itself to $R_2$, and then recursively to its contents, which consists currently of the new Security Goal $G_4$. Since $G_4$ is an EObject, its insertion will be part of the graph delta, as well as the insertion of all outgoing relations. The latter currently includes the redundancy requirement value assignment from $G_4$ to data value 2.

Altogether, the incremental pattern matcher will receive the graph delta $\delta = (\delta^\text{ent}, \delta^\text{rel}, \delta^+_\text{ent}, \delta^+_\text{rel})$, with $\delta^\text{ent} = \emptyset$, $\delta^\text{rel} = \emptyset$, $\delta^+_\text{ent} = \{+(G_4 :: \text{SecurityGoal})\}$, $\delta^+_\text{rel} = \{+(G_4 :: \text{redundancyRequirement}) \rightarrow 2\}$.

Finally, consider that if the redundancy requirement of $G_4$ is now changed from 2 to 1, the EObject $G_4$ emits an EMF change notification, which is likewise received by the newly attached
EContainmentAdapter. Since this attribute value update is a relation notification, it is interpreted directly as a relation insertion and a relation deletion, making the next graph delta \( \Delta = (\Delta_{\text{ent}}^-, \Delta_{\text{rel}}^-, \Delta_{\text{ent}}^+, \Delta_{\text{rel}}^+) \), with \( \Delta_{\text{ent}}^- = \emptyset \), \( \Delta_{\text{rel}}^- = \{- (G_4 \xrightarrow{\text{:(redundancyRequirement)}} 2)\} \), \( \Delta_{\text{ent}}^+ = \emptyset \), \( \Delta_{\text{rel}}^+ = \{ + (G_4 \xrightarrow{\text{:(redundancyRequirement)}} 1) \} \). The containment hierarchy is unchanged, so the adapter is not attached to or removed from any Notifiers.

### 5.4 Performance analysis of EMF model queries

This section will present a performance evaluation of EMF-IncQuery in a model validation scenario, taken from [11]. A further extensive set of benchmark results for EMF-IncQuery and comparison against several other query technologies is available at [ISR+13].

#### 5.4.1 Measurement scenario: constraint checking in AUTOSAR models

The EMF-IncQuery model query technique is demonstrated by checking well-formedness constraints over AUTOSAR [AUT] models (see Section 1.1.4.2).

To improve quality and reliability of electrical/electronic systems, the validation of AUTOSAR models should be carried out in the early stages of the development process. The standard specifies a multitude of constraints, which should be satisfied to ensure proper functionality in this diverse environment. In this measurement scenario, three of these constraints will be investigated.

#### 5.4.1.1 AUTOSAR core metamodel

![Ecore metamodel of basic AUTOSAR elements (extract)](image)

A simplified core part of the AUTOSAR [AUT] metamodel is shown in Figure 5.4. Every object in AUTOSAR inherits from the common ARObjec class. If an element has to be identified, it has to inherit from the Identifiable class, and the shortName attribute has to be set. AElement is a common base class for stand-alone elements, while specializations of FibexElement represent elementary building blocks within the FIBEX package. Instances of ARPackage class are arranged in a strict containment hierarchy by the subPackage association, and every PackageableElement can be aggregated by one of the ARPackages using the element association.
More specific subtypes will be introduced for the validation rules below.

5.4.1.2 ISignal constraint check

The first consistency check chosen from the AUTOSAR standard is the ISignal check, which is essentially a cardinality enforcement.

The two metamodel elements for this constraint (SystemSignal and ISignal) are illustrated in Figure 5.5, extending Figure 5.4. A SystemSignal is the smallest unit of data (it is unique per System) and it is characterized by its length (in bits). (Also two optional elements can be specified, Datatypes and DataPrototype constants, but they are not used in this example.) An ISignal must be created for each SystemSignal (these will be the signals of the Interaction Layer). Conversely, each ISignal must be associated with either a SystemSignal or a SystemSignalGroup. A signal group refers to a set of signals that must always be kept together to ensure the atomic transfer of information in them.

Figure 5.5: AUTOSAR metamodel extract (ISignal)

```
pattern CC_ISignal(isig)={
  ISignal(isig);
  neg ISignal.systemSignal(isig, _anySysSig);
}
```

Listing 5.4: Graph pattern for the ISignal consistency check

Listing 5.4 encodes the graph pattern CC_ISignal. The structural part contains only a single structural entity variable of type ISignal, but the NAC connects this entity to a SystemSignal instance via the ISignal.systemSignal association. Thus the graph pattern CC_ISignal matches ISignal instances that are not connected to a SystemSignal (or a group). This graph pattern can be used as a declarative model query, in order to validate the model against the structural well-formedness constraint that requires each ISignal to be connected to a SystemSignal.

5.4.1.3 Constraint check for system signal group (SSG) mapping

The second consistency check is significantly more complex than the previous one.

The required metamodel elements for this constraint check are likewise illustrated in Figure 5.5. A PDU (Protocol data unit) is the smallest information which is delivered through a network layer. It is an abstract element in AUTOSAR, and has multiple different subtypes according to the available network layers. This case study will only examine IPdu (Interaction Layer PDU), particularly
5.4. PERFORMANCE ANALYSIS OF EMF MODEL QUERIES

SignalIPdu elements. These SignalIPdus are used to transfer ISignals. The positions of these ISignals are defined by the ISignalToIPduMappings. As discussed before, ISignal are associated with either a SystemSignal or a SystemSignalGroup.

To ensure the atomic transfer of a SystemSignalGroup, they have to be packed properly into SignalIPdus. This means that if a SystemSignalGroup is referenced from a given SignalIPdu (via an ISignalToIPduMapping), then every SystemSignal in it should be referenced as well from that SignalIPdu (note that an ISignalToIPduMapping references ISignals, but as every SystemSignal and SystemSignalGroup must have an ISignal, this is not a problem – the parent-child relationship is thus expressed between the SystemSignal and SystemSignalGroup instances). Conversely, if a SystemSignal is mapped to an SignalIPdu, then its parent SystemSignalGroup must be mapped to it as well.

This latter constraint is formulated as graph pattern CC_SystemSignal (shown in Listing 5.5 along with helper patterns), matching one of the two cases of violation where the mapping element corresponding to the SystemSignalGroup is missing (as indicated by the NEG condition), even though the child SystemSignal is mapped.

5.4.1.4 Simple Channel consistency check

To demonstrate the third consistency check, some additional AUTOSAR elements have to be described. These elements are illustrated by Figure 5.6, extending Figure 5.4.

In AUTOSAR, ECU (Electronic Control Unit) instances can communicate with each other through a communication medium represented by a PhysicalChannel. Physical Channels are aggregated by a CommunicationCluster, which is the main element to describe the topological connection of communicating ECUs. A Physical Channel can contain ISignalTriggering and IPduTriggering elements. The IPduTriggering and ISignalTriggering describe the usage of IPdus and Signals on physical channels. ISignalTriggering defines the manner of triggering of an ISignal on the channel, on which it is sent. IPduTriggering describes on which channel the IPdu is transmitted.

The following constraint has to be satisfied for a physical channel: if a PhysicalChannel chan contains a SignalIPdu iPdu (through an IPduTriggering), then each ISignal iSig that is contained by iPdu (through an ISignalToIPduMapping) must have a related ISignalTriggering in the channel chan. In other words the channel is invalid if there is at least one ISignal iSig that has no related

```
pattern CC_SystemSignal(mChild, isParent, ssParent) = {
  find systemChild(isChild, ssChild, ssParent, isParent);
  find signalOfPDU(pdu, mChild, isChild);
  neg find signalOfPDU(pdu, mParent, isParent);
}

pattern systemChild(isChild, ssChild, ssParent, isParent) = {
  SystemSignalGroup.systemSignal(ssParent, ssChild);
  ISignal.systemSignal(isChild, ssChild);
  ISignal.systemSignal(isParent, ssParent);
}

pattern signalOfPDU(pdu, map, iSig) = {
  SignalIPdu.signalToPduMapping(pdu, map);
  ISignalToIPduMapping.signal(map, iSig);
}
```

Listing 5.5: Pattern to find invalid signal group mappings
CHAPTER 5. INCREMENTAL MODEL QUERIES OVER INDUSTRIAL EMF MODELS

Figure 5.6: AUTOSAR metamodel extract (Channel)

I.SignalTriggering in the channel. This informal definition is formalized in Listing 5.6 as a graph pattern. If pattern CC_Channel(chan) can be matched for a Physical channel chan, then it is considered to be invalid.

```
pattern CC_Channel(chan) = {
  PhysicalChannel.iPduTriggering.iPdu(chan, iPdu);
  SignalIPdu.signalToPduMapping.signal(iPdu, iSig);
  neg PhysicalChannel.iSignalTriggering.signal(chan, iSig);
}
```

Listing 5.6: Pattern to find invalid physical channels

5.4.2 Benchmarking

The benchmark simulates the typical scenario of model validation. The user is working with a large model, the modifications are small and local, but the result of the validation needs to be computed as fast as possible. To emulate this, the benchmark sequence consists of the following sequence of operations:

1. First, the model is loaded into memory. In the case of EMF-IncQuery, most of the overhead is expected to be registered in this phase, as the pattern matcher cache needs to be constructed. Note however, that this is a one-time penalty, meaning that the cache will be maintained incrementally as long as the model is kept in memory.

2. Next, in the first query phase, the entire match set of the constraints is queried. This means that a complete validation is performed on the model, looking for all elements for which the constraint is violated.

3. After the first query, model manipulations are executed. These operations only affect a small fixed subset of elements, but change the constraint’s validity.

4. Finally, in the second query phase, the complete validation is performed again, to check the net effect of the manipulation operations on the model.
In addition to the EMF-IncQUERY-based implementation, two alternative prototypes were involved in the performance comparison: a plain Java variant and an OCL variant that uses MDT-OCL[Ecl11]; neither of which apply incremental techniques.

The benchmark models were generated automatically to be large enough for performance measurements, and at the same time contain some violations of the constraints in the case study. The simulated model editing performed in Step 3 is a relatively short automated manipulation sequence that was designed to cause new violations or repair existing violations. See [11] for details on the generated instance models and the applied model modifications.

The measurement was carried out in 2010. The exact versions of EMF and MDT-OCL were 2.5.0 and 1.2.0 respectively, running on Eclipse Galileo SR1 20090920-1017. The benchmarks ran on an Intel Core2 E8400-based PC clocked at 3.00GHz with 3.25GBs of RAM on Windows XP SP3 (32 bit), using the Sun JDK version 1.6.0_17 (with a maximum heap size of 1536 MBs). Execution times were recorded using the java.lang.System class, while memory usage data has been recorded in separate runs using the java.lang.Runtime class (with several garbage collector invocations to minimize the transient effects of Java memory management). The data shown in the results correspond to the averages of 10 runs each.

All implementations share the same code for model manipulation. They differ only in the query phases:

- The EMF-IncQUERY variant uses the EMF-IncQUERY for reading the match set of the graph patterns corresponding to constraints. These operations are only dependent on the size of the graph pattern and the size of the matching set itself (this is empirically confirmed by the results, see Section 5.4.3). To better reflect memory consumption, the RETE nets for all three constraints were built in each case.

- The plain Java variant performs model traversal using the generated model API of EMF. This approach is not naive, but intuitively manually optimized based on the constraint itself (but not on the actual structure of the model [14]).

- The OCL variant has been created by systematically mapping the contents of the graph patterns to OCL concepts, to ensure equivalence. We did not perform any OCL-specific optimization.

To ensure the correctness of the Java implementation, a set of small test models was created in order to verify the results manually. The rest of the implementations have been checked against the Java variant as the reference, by comparing the number of valid and invalid matches found in each round. See [11] for more information about the exact EMF-IncQUERY queries, OCL expressions and Java code used for the benchmarks.

5.4.3 Analysis of the results

Based on the results (Table 5.1), we have made the following observations:

1. As expected, query operations with EMF-IncQUERY are nearly instantaneous, they are only measurable for larger models (where the match set itself is large, and it takes a considerable amount of time just to go through all matches). In contrast, both Java and OCL variants exhibit a polynomially increasing characteristic, with respect to model size. The optimized Java implementation outperforms OCL, but only by a constant multiplier.
2. Although not shown in Table 5.1, the times for model manipulation operations were also measured for all variants, and found to be uniformly negligible. This is expected since very few elements are affected by these operations, therefore the update overhead induced by the Rete net is negligible.

3. The major overhead of EMF-InCQuery is registered in the resource loading times (shown in the Res column in Table 5.1). It is important to note that the loading times for EMF itself is included in the values for EMF-InCQuery. By looking at the values for loading times and their trends, it can be concluded that EMF-InCQuery exhibits a linear time increase in both benchmark types, with a factor of approximately 2 compared to the pure EMF implementation. MDT-OCL does not cause a significant increase.

4. The memory overhead also grows linearly with the model size, but depends on the complexity of the constraint, too. More precisely, it depends on the size of the match sets of patterns and that of some sub-patterns depending on the structure of the constructed Rete network. (Actually, the memory overhead is sub-additive with respect to patterns, due to a varying degree of Rete node-sharing.)

It has to be emphasized that in practical operations, the resource loading time increase may not be important as it occurs only once during a model editing session. So, as long as there is enough memory, EMF-InCQuery provides nearly instantaneous query performance, independently of the complexity of the query and the contents of the model. In certain cases with complex queries, like for the SSG benchmark, EMF-InCQuery is the only variant where the query can be executed in the acceptable time range for large models above 500 000 elements, even when we take the combined
times for resource loading and query execution into consideration. The performance advantage is less striking in other cases, as indicated by the figures for the Channel and ISignal benchmarks, where the difference remains in the range of a few seconds even for large models.

Overall, EMF-InCQUERY suits application scenarios with complex queries, which are invoked many times, with relatively small model manipulations in-between. Even though the memory consumption overhead is acceptable even for large models on today’s PCs, the optimization techniques based on combining various pattern matching techniques [14] previously presented for ViATRA2 apply to EMF-InCQUERY too (even if their implementation over EMF will require some future work).

### 5.5 Chapter conclusions

The chapter presented EMF-InCQUERY as the next evolutionary step in efficiently executing complex queries over EMF models by adapting incremental graph pattern matching technology.

The proposed query language syntax is derived from the graph pattern fragment of the ViATRA2 transformation language [VB07] and tailored to the task of querying EMF models, with additional significant semantic extensions to its predecessor. The execution mechanism reuses the core concepts of Rete networks from my previous results and integrates it to EMF. Measurements have confirmed that the technique provides fast evaluation of complex model queries.
Chapter 6

Queries and Transformations for Security Requirements

This chapter partially follows [18] and [33].

6.1 Introduction

Modern software systems are increasingly complex and the environments where they operate are increasingly dynamic. The number and needs of stakeholders are also changing constantly as they adjust to changing environments. A consequence of this trend is that the requirements for a software system are numerous and they change continuously. To deal with evolution, we need analysis techniques that assess the impact of system evolution on the satisfaction of requirements. Requirements for system security, in particular, are very sensitive to evolution: security properties that are satisfied before the evolution might no longer hold or as result of the evolution.

Another important aspect is the change management process itself which is a major problem in practice. Changes make the traceability of requirements difficult and the monitoring of requirements unreliable: requirements management is time-consuming and error-prone when done manually. Thus, a semi-automated requirements evolution management environment, supported by a tool, will improve requirement management with respect to keeping requirements traceability consistent, realizing reliable requirements monitoring, improving the quality of the documentation, and reducing the manual effort.

Section 6.1.1 presents SeCMER [18], a tool developed in the context of the SecureChange European project [EU 12]. The tool supports the different steps of SeCMER methodology for evolutionary requirements [31].

The SeCMER tool provides an opportunity for leveraging a synergy of novel techniques introduced throughout the thesis. Change-driven transformations (addressed in Chapter 7) with incremental pattern matching (addressed in Chapter 3) are employed on an industrial EMF platform (addressed in Chapter 5) to ensure change propagation (see Section 6.4) monitor argument validity (see Section 6.3), to automatically detect violations or fulfillment of security properties (see Section 6.2), and to issue alerts prompting human intervention, a manual analysis or argumentation process, or trigger automated reactions in certain cases.
6.1.1 Overview of the SéCMER tool

The goal of the SéCMER tool is to support the requirement engineers in following the associated SéCMER methodology. During the entire activity of requirements engineering, requirements model can undergo automated, pattern-based static analysis and manual, informal argumentation analysis to discover security issues.

The tool provides basic viewing and editing functionality:

- Requirements elicitation: editing the security requirements model to identify e.g. stakeholders, resources provided, actions performed, functional and security goals stated and met, decomposition, input / output dependencies of actions or goals, trust between stakeholders and finally delegation of duties or access. Requirements models are represented internally in the SéCMER conceptual model for security requirements (see [MMP+11] or Section 5.1.2.1 for a simplified extract). Instead of displaying the abstract model, however, the user interface uses a modified version of the Si* / Tropos visual syntax [MMZ07].

- Recording arguments carried out by security experts that identify security liabilities and problems. Also modeling the breakdown structure of arguments, the interrelation with counter-arguments (rebuts, mitigations), and the back-tracing of elementary facts to concepts in the requirements model. See [TYHN10] to learn more.

Additionally, the following added-value mechanisms are implemented:

- An automated static security analysis considers a class of security problems that are defined by an extensible set of security constraints. Each security constraint is a graph pattern-based model query that identifies parts in the model that violate a given security property. The analysis detect violations of the given security properties and offer automated solutions. Violations appear as Eclipse problem markers (warnings). The suggested solutions appear as Quick Fix rules. For details, see Section 6.2.

- Traceability links (“evidence”) can be established between the argument and requirement models. They enable automatic detection of requirement changes that make a manually conducted argument obsolete. Model changes involving the ground facts may trigger a notification that alerts the user about the possibility that the argument may have become invalid due to the change. The security experts can then revisit these arguments to reflect the evolution, while no costly revision process is required for unaffected arguments. For details, see Section 6.3.

- The requirements model is always represented as a visual Si*/Tropos diagram, and also as a more abstract underlying SéCMER model. There is on-the-fly bi-directional synchronization between the abstract SéCMER requirement model and its Si* concrete syntax representation. Changes made in either model, whether initiated by the editor functionality or e.g. via a Quick Fix, are transformed and synchronized between the SéCMER abstract syntax and Si* aspect on the fly. For details, see Section 6.4.

A screencast\(^1\) demonstrates the SéCMER tool in action, via a feature tour that presents various scenarios including most of the examples that will be used throughout Section 6.1.

\(^1\)http://www.youtube.com/watch?v=OywZeNeSuJM
6.1.2 Metamodels in the S/e.scCMER tool

As follows from Section 6.1.1, the tool integrates heterogeneous models conforming to different metamodels. The abstract security requirement model conforms to metamodel $MM_{S/e.scCMER}$, a simplified extract of which has already been introduced in Section 5.1.2 on page 97.

6.1.2.1 Concrete syntax: Si*

The concrete syntax of the requirement model is based on the Si*/Tropos diagram, thus there is also an Si* model conforming to metamodel $MM_{Si*}$. Section 6.4 will provide more information on $MM_{Si*}$.

The S/e.scCMER tool provides bidirectional synchronization between the abstract and concrete representations. Most traditional approaches for implementing bidirectional synchronization require a correspondence model between them conforming to $MM_{corr}$, and also an unidirectional glue (see Section 2.1.3) for each of $MM_{S/e.scCMER}$ and $MM_{Si*}$ that points from $MM_{corr}$ to the respective requirements formalism: $MM_{S/e.scCMER} \xleftarrow{MM_{glue1}} MM_{corr} \xrightarrow{MM_{glue2}} MM_{Si*}$. Note however, that Section 7.3 will present a way to define the bidirectional synchronizing transformation with no need for such correspondence models whatsoever.

6.1.2.2 Argumentation support

The goal of the argument models is to support or refute the satisfaction of security requirements. The arguments are recorded in a structured fashion, so that overall conclusions are drawn from more basic statements, which may themselves be the consequence of earlier claims; this tree of arguments is ultimately based on assumptions and ground facts. Some of these facts originate from the security requirement model, and will be called here evidence.

The actual argument model has rich structure. Yet for the purposes of the case study in this thesis, the argument metamodel $MM_{arg}$ can be simplified to a single class Argument. The case study metamodel is then obtained by the glue-merge $MM_{arg} \xrightarrow{MM_{evidence}} MM_{S/e.scCMER}$ of this argument metamodel to the security requirement metamodel (see Figure 5.2) along the single glue association evidence, which is pointing from Argument of the argumentation metamodel to Requirement Entity of the security requirement metamodel.

6.1.3 Example scenarios from the ATM domain

The features supported by the S/e.scCMER tool will be illustrated using one of the case studies of the SecureChange research project [EU 12], the ongoing evolution of ATM (Air Traffic Management) systems as planned by the ATM 2000+ Strategic Agenda [ATM03] and the SESAR Initiative. Section 5.1.2 on page 97 provides an example scenario of security requirements unrelated to the ATM domain.

Part of ATM system’s evolution process is the introduction of the Arrival Manager (AMAN), which is an aircraft arrival sequencing tool to help manage and better organize the air traffic flow in the approach phase. The introduction of the AMAN requires new operational procedures and functions.

It is necessary to preserve specific security properties after the deployment of the identified changes. In particular, an operational need-to-know principle can be defined in terms of the following security properties:

**Information Access.** Authorized actors must have access to confidential information regarding queue management in the terminal area. Access to information needs to comply with specific role-based access control rules drawn from the operational requirements.
CHAPTER 6. QUERIES AND TRANSFORMATIONS FOR SECURITY REQUIREMENTS

Information Protection. Unauthorized actors are not allowed to access confidential queue management information.

Information Need. Confidential queue management information can be accessed by authorized actors only when the information is necessary for operational purposes, which may vary even in real time, due to particular conditions (bad weather, emergency status, etc.).

6.1.3.1 Scenario 1: trusted path violation

The new, electronic administration and management functions of AMAN are supported by a new information management system for the whole ATM, an IP based data transport network called System Wide Information Management (SWIM) that will replace the current point to point communication systems with a ground/ground data sharing network which connects all the principal actors involved in the Airport Management and the Area Control Centers.

The actors involved in the simple scenario are the AMAN, the Meteo Data Center (MDC), the SWIM-Box and the SWIM-Network. The SWIM-Box is a terminal of the SWIM information management system which provides access via defined services to data that belong to different domain such as flight, surveillance, meteo, etc. The introduction of the SWIM poses new questions of security, as sensitive data assets may be routed through components of a centralized network.

Figure 6.1(a) shows the pre-state requirement model in the Si* concrete syntax (the image is an actual screenshot from the SeCMER tool). The requirement model contains two actors: the AMAN and MDC, both depicted as red circles, each with a so-called sphere of influence. MDC provides the asset MeteoData and delegates (communicates) it to the AMAN. In the Si* syntax, this is expressed by MeteoData being present in the sphere of influence of MDC, and being delegated ("Dp" arrow) from there into the sphere of influence of AMAN. The AMAN has an integrity security goal MDIntegrity for MeteoData, likewise indicated by showing the goal in the sphere of influence of the actor. MDC is entrusted ("Te" arrow) to comply with this security goal, i.e. it will provide correct weather information. AMAN also performs a Task, SecurityScreening (once again indicated by the sphere of influence), to regularly conduct a background check on its employees to ensure that they do not expose to risk the information processed by the AMAN.

As the evolution of the system introduces SWIM to mediate the communication between the AMAN and MDC, the model evolves as follows (see Figure 6.1(b)):

- The Actors SWIM, SWIMBox_MDC and SWIMBox_AMAN are introduced in the Si* model
- As the meteo data is no longer directly provided by MDC to AMAN, the delegation relation between the two is removed.
- Delegation relationships are established between the Actors MDC, SWIMBox_MDC, SWIM, SWIMBox_AMAN, AMAN, according to the route of the MeteoData data resource.
- As the SWIM network can be accessed by multiple parties, the AMAN has a new security goal MDAccessControl protecting the MeteoData asset.

6.1.3.2 Scenario 2: least privilege violation

The introduction of the Arrival Management subsystem (AMAN) affects Controller Working Positions (CWPs) as well as the Area Control Center (ACC) environment as a whole. The main foreseen change in the ACC from an operational and organizational point of view is the automation of tasks (i.e. the
6.1. INTRODUCTION

(a) Evolution pre-state

(b) Evolution post-state (changes highlighted)

Figure 6.1: Scenario 1 evolution as Si’ diagram
usage of the AMAN for the computation of the Arrival Sequence) that in advance were carried out by Air Traffic Controllers (ATCOs), a major involvement of the ATCOs of the upstream Sectors in the management of the inbound traffic.

See Figure 6.2 for the pre-state of the model, and Figure 6.3 for the post-state after the introduction of AMAN (both images are actual screenshots from the SeCMER tool, with change highlighting added to the second one).

Among other duties, the AMAN schedules the arrival of a State Flight, which is a highly sensitive flight with high-ranking state officials on board. In the post-state of the evolution, due to the unified electronic representation of flights required by AMAN, information regarding the State Flight must be handled by the ACC systems as part of the Flight Data.

ATCOs that are currently on duty in the ACC control room are aware of this sensitive information and take that into account while working on their traffic control sectors. The ATCO supervisor, among other goals, is responsible for the security of confidential information. Each operation CWP shows any relevant information about the flights. It is necessary to guarantee that confidential information becomes available to actors operating inside and outside the ACC control room only when information is necessary for achieving their operational goals. For instance, an external contractor’s System Engineer who is authorized to access the control room and a CWP to perform system maintenance should not be allowed to access confidential information such as the State Flight.

6.1.3.3 Scenario 3: argumentation

The following scenario will involve a recorded argument and a linked security requirement model that provides evidence to the argument. As the arguments and especially the evidence relations are not visually depicted on the Si* diagram, the scenario is illustrated by a non-standard visual syntax of the abstract SeCMER requirement model (similarly to Section 5.1.2 on page 97) and its relationship with the relevant argument.

Sample model Figure 6.4(a) shows a security model for air traffic communication systems. The three actors are Air Traffic Controller (ATCO), Airlines (AL) and Catering Services (CS). Actor ATCO provides resource RAS (runway assignment), and delegates (i.e. communicates) it to AL. Actor AL provides resource MO (meal orders), and delegates it to CS.

The integrity of data asset RAS is security-critical, because if terrorists were able to change RAS, they could make planes crash. To ensure the integrity of RAS, ATCO carries out the following Tasks:

- \( T_1 \) “Use data security and secure communication technologies”, to make sure the Air Traffic Controller is in control of RAS.
- \( T_2 \) “Conduct a yearly IT security training of the whole staff”, to reduce the likelihood of social engineering attacks.
- \( T_3 \) “Enforce policy that every manual decision has to be approved by a second member of the controller staff”, to reduce the impact of human error or malice.
- \( T_4 \) “Perform a quarterly security screening of employees”, monitoring whether an employee has big debts, or can be blackmailed into helping criminals, or has befriended terrorists, etc. to reduce the likelihood of malice.

To decide whether the integrity of RAS will be maintained, security experts conduct an informal argumentation analysis \( ARG \). Based on the model and their background knowledge, they judge that
Figure 6.2: Scenario 2 evolution pre-state as Si* diagram
Figure 6.3: Scenario 2 evolution post-state-state as Si* diagram (changes highlighted)
6.1. INTRODUCTION

(a) Evolution pre-state

(b) Post-state of an evolution not requiring re-evaluation (changes highlighted)

Figure 6.4: Scenario 3: link between a SèCMER security model and an argument

they are confident in this security requirement. Tasks $T_{1-4}$ and Resource $RAS$ are used in their argument, so they are recorded in the model as the evidences for $ARG$.

An evolution triggering re-evaluation A possible evolution of the model is the following: to cut back costs, $ATCO$ plans to reduce the frequency of security screenings, so $T_4$ will be modified to "Perform a yearly security screening of employees". Thus change is inflicted on an evidence for $ARG$. Since we have no formal way to determine whether the modified Task can fulfill the security needs, the argumentation experts must be alerted to revisit argument $ARG$. They will then decide e.g. that the security requirement is still met with the weakened guarantees, based on regulation, previous experience and a risk analysis conducted by Risk Engineers.

An evolution not requiring re-evaluation Another possible evolution is that the communication between $AL$ and $CS$ is now routed through a new Actor $SWIM$ (System-Wide Information Management), as shown in Figure 6.4(b). This change can have a wide influence on the system, but it does
not invalidate the argument \( ARG \), as no evidence of \( ARG \) was involved in the change. Therefore this time there is no need for the argumentation experts to exert further manual effort.

### 6.2 Continuous validation of security requirements models

The SeCMER methodology includes a lightweight automated analysis step that evaluates requirements-level compliance with security principles. These security principles are declaratively specified by an extensible set of security constraints.

A security constraint expresses a situation (a graph-like configuration of model elements) that leads to the violation of a security property. Whenever a new match of the security constraint (i.e. a new violation of the security property) emerges in the model, it can be automatically detected and reported. The specification of security constraints may also be augmented by automatic remedies (i.e. templates of corrective actions) that can be applied in case of a violation to fix the model and satisfy the security property once again.

The efficient and continuous validation of the EMF-based requirement models is provided by the incremental matching functionality of EMF-InCQ\(\)U\(\)R\(\)Y (see Chapter 5). Incrementally evaluated queries indicate violations of these security constraints, which will appear as problem markers (warnings). The suggested solutions appear as Quick Fix rules offered for the problem marker.

SeCMER includes extension facilities that allow plug-ins to contribute the declarative definition of security constraints in the high-level model query language of EMF-InCQ\(\)U\(\)R\(\)Y (see Section 5.2). Automated solution templates (defined programmatically) can also be contributed.

Although the set of security constraints is extensible, the tool prototype is delivered with a default set that should be suitable for most application domains of security requirements engineering. The main focus points of these security constraints are the following concepts: trust (which can be explicitly modeled, and interpreted transitively), access (which can also be granted/delegated transitively), and need (expressed by carrying out an action that consumes a resource). The patterns are further characterised by the following:

- The security constraints only consider assets that are protected by security goals.
- If a trusted actor performs an action that is known to fulfill the security goal, then no further investigation of that goal is required.
- If an actor has access to an asset without trust (regardless of need), then it is considered a violation of the trusted path security constraint.
- If an actor has access to an asset without the need thereof (regardless of trust), then it is considered a violation of the least privilege security constraint.
- The above security violation reports can be suppressed by manual arguments supporting the satisfaction of the security goal.
- If an actor has need for an asset but no actual access to it, then the model is reported as inconsistent/incomplete.

**Example 39** For instance, the trusted path security constraint introduced in Section 5.1.2.2 on page 98, finds security violations where an asset is communicated via an untrusted path. The pattern has the following structure: if a concerned actor wants a security goal that expresses that a
6.2. CONTINUOUS VALIDATION OF SECURITY REQUIREMENTS MODELS

(a) Detected security violations in Scenario 1

![Image](path/to/figure6.5a.png)

(b) Detected security violations in Scenario 2

![Image](path/to/figure6.5b.png)

Figure 6.5: Detection of security issues

Detailed definition of other security and consistency constraints enforced by the SeCMER tool are omitted here for brevity.

**Example 40** In Scenario 1 (see Section 6.1.3.1), the integrity property for MD is violated because AMAN entrusts MDC with the integrity security goal, but not the intermediary actors SWIM-Box_MDC, SWIM and SWIMBox_AMAN. The violation (i.e. a match of the pattern from Listing 5.1) is detected and reported by the tool, as shown on Figure 6.5(a).

**Example 41** In Scenario 2 (see Section 6.1.3.2), Actor ACC Supervisor provides State Flight Info and wants to have its confidentiality preserved. In the post-state, the asset forms a part of the aggregate resource Flight Data, which is accessible to various Control Room actors (e.g. via CWP), eventually including the System Engineer. Although System Engineer is trusted by the ACC Supervisor with the confidentiality security goal, there is no actual need for the former to have access to State Flight Info, as CWP maintenance only requires access to CWP Software, not the whole CWP containing valuable data assets. Therefore this is a violation of the least privilege security constraint, and the tool marks this situation as such (see Figure 6.5(b)). The situation could be resolved either by carefully restricting the access privileges of System Engineer (also making sure that the new restrictions do not interfere with normal duties of the actor); or alternatively by supplying an informal argumentation that states that this specific case causes little security risk, and marking the Argument as supportive of the confidentiality security goal to suppress the warning.

Assuming that a graph pattern finds violations of security constraints, requirements engineers are further assisted by a set of suggested fixes that can be applied on violations of the security property; the set of these Quick Fix suggestions is also extensible. The requirements engineers can then choose one of the suggestions, or come up with their own solution. In the terminology of Section 2.1.5, the Quick Fixes are model manipulation operations parametrized by a match of a graph pattern corresponding to a security constraint.

In case of the trusted path security constraint as formulated in Listing 5.1, possible examples of corrective actions include:
• Add a trust relationship from concernedActor to untrustedActor to reflect that the security decision was that there must be trust between these actors (e.g. by establishing a liability contract between them).

• Alternatively, a task can be created that explicitly fulfills seeGoal, such as introducing a policy or technological process that makes it impossible for untrustedActor to abuse the situation (e.g. digital signature to ensure the security goal of data integrity). The task must be performed by concernedActor or someone trusted by concernedActor.

These solution templates can be attached to the security constraint so that they are offered whenever a violation of the corresponding security property is detected. The solutions can be implemented by arbitrary program code, typically short Java snippets that manipulate the model according to the description of the solution.

Example 42 In case of Scenario 1, the suggested Quick Fixes for the specific violation related to the SWIM actor are shown in Figure 6.6.

![Figure 6.6: Suggested corrective actions (Scenario 1)](image)

6.3 Change impact analysis on informal arguments

Security requirement models have their set of well-formedness and security constraints, ensuring that the model is meaningful, consistent and secure. Graph pattern-based on-the-fly validation of such constraints was already addressed earlier. These constraints are static in the sense that they only restrict the current state of the model. However, there are cases where evolutionary constraints are needed, that can take into account how the model changes.

In the example evolutions outlined in Scenario 3 of Section 6.1.3.3, invalidating informal arguments was such a problem. Formal and informal argumentation is carried out using the requirement model, to determine which security requirements are met. This argumentation is a laborious and
6.4. BIDIRECTIONAL CHANGE-DRIVEN REQUIREMENTS SYNCHRONIZATION

costly process requiring significant human expertise. In evolving security-critical applications, it is important that the argumentation is only revised for those security requirements that are influenced by the change of the model. Thanks to the traceability from Argument to evidence, there is enough information to determine which arguments need to be re-evaluated.

The argument has to be invalidated if one of its evidences is involved in a change. This, however, cannot directly be determined by using only the present (post-state) of the security model. The trivial reason for this is that there could have been various histories leading up to the same post-state; revisiting the arguments is only needed in some of the cases, but static constraints cannot distinguish them.

Since supporting evolving systems is a major goal of S/e.scCMER, the argumentation phase has to deal with change as well; therefore the tool provides enforcement of this evolutionary constraint. Constructing such an invalidation mechanism is simplified by the change-driven transformation formalism that will be proposed in Chapter 7. See Section 7.4 for a solution demonstrating the applicability of change-driven techniques to this problem.

6.4 Bidirectional change-driven requirements synchronization

Internally, the S/e.scCMER tool represents requirements in a model conforming to the S/e.scCMER requirements metamodel [MMP⁺11], a heavily simplified extract of which is shown in Section 5.1.2. The user interface, however, features a graphical editor using the Si* concrete syntax [MMZ07], which is a well-known type of requirements modeling diagram.

The S/e.scCMER tool maintains a bidirectional synchronizing live transformation between Si* and the S/e.scCMER model, in analogy to bidirectional transformations between abstract syntax and concrete syntax graphs of other modeling languages [RÖV10]. This way, changes made to the Si* diagram via the user interface are immediately propagated to the abstract syntax conforming to the S/e.scCMER conceptual metamodel, and changes made directly to the abstract syntax (e.g. via invoking a Quick Fix) are likewise propagated to the Si* concrete syntax.

The following sections give an outline of the most important challenges of this transformation, without presenting a solution. Constructing such a solution is greatly simplified by change-driven transformation specification, for which a language will be proposed in Chapter 7. Based on that language, Section 7.3 will showcase some interesting parts of a transformation specification that satisfies the bidirectional live synchronization requirements proposed here. The solution will also demonstrate how the change-driven technique is an appropriate answer for the specific challenges of this bidirectional live synchronization task.

6.4.1 Properties of the Si* metamodel

The Si* (short for Secure i*) diagram notation is based on i* [Yu96]. Its purpose is to represent an actor-goal model of security requirements, and has therefore more or less similar elements as the S/e.scCMER metamodel. Between the two metamodels, there are also some slight deviations in terminology, structure and the refinement of certain types of information (e.g. an Actor can be either a Stakeholder or an Agent); these are however uninteresting from the point of view of the thesis and will not be discussed further.

There is, however, a significant characteristic of the version of Si* metamodel integrated into the S/e.scCMER tool that has a large influence on the transformation. While the S/e.scMER Requirement Entity has a unique name attribute (meaning that no two of them have the same name), the name of Si* elements is not unique. For example, each Actor may appear multiple times on a Si* diagram; this
helps in constructing large requirement models as the sphere of influence of the actor does not have to be contiguous. Similarly, other kinds of modeling elements may also appear multiple times on an Si* diagram, such as the two occurrences of MeteoData in Figure 6.1(a). As a perhaps surprising design choice of the originals creators of the Si* metamodel, this duplication would actually correspond to multiple elements in the Si* model that share a common name and are otherwise unlinked. From the point of view of the transformation, the entire collection of Si* elements sharing a name would correspond to a single SnCMER model element with the same name.

### 6.4.2 Mapping between the languages

As the two modeling languages have slightly different expressive power and different structure, therefore the following challenges arise in the model-to-model synchronization:

1. some concepts are not mapped from one formalism to the other or vice versa,
2. some model elements may be mapped into multiple (even an unbounded amount of) corresponding model elements in the other formalism, and finally
3. it is possible that a single model element has multiple possible translations (due to the source formalism being more abstract); one of them is created as a default choice, but the other options are also accepted.

<table>
<thead>
<tr>
<th>Requirement name and multiplicity</th>
<th>Si* elements</th>
<th>SnCMER elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 Actor2Actor ':1'</td>
<td>Actors of the same name</td>
<td>Actor</td>
</tr>
<tr>
<td>R2 Resource2Resource ':1'</td>
<td>Resources of the same name</td>
<td>Resource that is provided</td>
</tr>
<tr>
<td>R3 Resource2provides ':1'</td>
<td>original copy of Resource, owned by an Actor and not received through delegation</td>
<td>provides relation from Actor to Resource</td>
</tr>
<tr>
<td>R4 Resource2consumes ':1'</td>
<td>additional copy of Resource, received by an Actor through delegation and not delegated further</td>
<td>consumes relation from Actor to Resource</td>
</tr>
<tr>
<td>R5 Task2Task ':1'</td>
<td>Tasks of the same name</td>
<td>Task that has incoming does relation</td>
</tr>
<tr>
<td>R6 Task2does ':1'</td>
<td>final copy of Task, owned by an Actor and not delegated further</td>
<td>does relation from Actor to Task</td>
</tr>
<tr>
<td>R7 SoftGoal2Goal ':1'</td>
<td>Softgoals of the same name</td>
<td>Security Goal that is wanted</td>
</tr>
<tr>
<td>R9 Goal2wants ':1'</td>
<td>original copy of a Si* Goal or Softgoal, owned by an Actor and not received through delegation</td>
<td>wants relation from Actor to Goal</td>
</tr>
<tr>
<td>R10 AND2AND ':1'</td>
<td>AND Composition</td>
<td>And Decomposition between tasks or between goals, if both endpoints are mapped</td>
</tr>
<tr>
<td>R11 OR2OR ':1'</td>
<td>OR Composition</td>
<td>Or Decomposition between tasks or between goals, if both endpoints are mapped</td>
</tr>
<tr>
<td>R12 MeansEnd2fulfills ':1'</td>
<td>MeansEnd relation from a Task to a Goal or Softgoal</td>
<td>fulfills relation between Task and Goal, if both endpoints are mapped</td>
</tr>
<tr>
<td>R13 MeansEnd2consumes ':1'</td>
<td>MeansEnd relation from a Resource to a Task or Goal</td>
<td>consumes relation between Task / Requirement and Resource, if both endpoints are mapped</td>
</tr>
<tr>
<td>R14 MeansEnd2produces ':1'</td>
<td>MeansEnd relation from a Task or Goal to a Resource</td>
<td>produces relation between Task / Requirement and Resource, if both endpoints are mapped</td>
</tr>
<tr>
<td>R15 Custom2protects ':1'</td>
<td>Custom relation from a Softgoal to a Resource / Task</td>
<td>protects relation between Security Goal and asset, if both endpoints are mapped</td>
</tr>
<tr>
<td>R16 De2Delegation ':1'</td>
<td>Delegation of Permission, pointing from a Resource owned by a &quot;delegator&quot; Actor to a &quot;receiver&quot; Actor</td>
<td>Delegation with a delegator and a receiver Actor and Resource as dependum, if all three endpoints are mapped</td>
</tr>
<tr>
<td>R17 De2Trust ':1'</td>
<td>Delegation of Permission, pointing from a Task or Goal owned by a &quot;delegator&quot; Actor to a &quot;receiver&quot; Actor</td>
<td>Delegation with a delegator and a receiver Actor and Task or Goal as dependum, if all three endpoints are mapped</td>
</tr>
<tr>
<td>R18 Tip2Trust ':1'</td>
<td>Trust of Permission, pointing from a Resource owned by a &quot;trustee&quot; Actor to a &quot;trusted&quot; Actor</td>
<td>Trust with a trustee and trusted Actor and Resource as dependum, if all three endpoints are mapped</td>
</tr>
<tr>
<td>R19 Te2Trust ':1'</td>
<td>Trust of Execution, pointing from a Task or Goal owned by a &quot;trustee&quot; Actor to a &quot;trusted&quot; Actor</td>
<td>Trust with a trustee and trusted Actor and Task or Goal as dependum, if all three endpoints are mapped</td>
</tr>
</tbody>
</table>

Table 6.1: Transformation requirements
6.5 Chapter conclusions

I have designed an integrated environment for security requirement engineering. The capabilities of the tool include validating security properties of the model, seamlessly representing the knowledge in two modeling formalisms using bidirectional live transformation, and minimizing the impact of changes on security arguments.

Model validation can be carried out using incremental model queries, as shown before. Actually implementing the transformation in a live and bidirectional way, however, poses a number of technical and conceptual challenges, and so does the realization of change impact analysis. Chapter 7 will present my contributions towards change-driven transformations, providing a solution to these issues.
In the chapter, I investigate change-driven model transformations, which are directly defined to be triggered by complex model changes carried out by arbitrary transactions on the model (e.g., editing operation, transformation, etc).

After a classification of relevant change scenarios, challenges are identified for change-driven transformations. As the main technical contribution of the chapter, an expressive, high-level language is defined for specifying change-driven transformations as an extension of graph patterns and graph transformation rules. This language generalizes previous results on live model transformations by offering trigger events for arbitrarily complex model changes, and dedicated reactions for specific kinds of changes, making this way the concept of change to be a first-class citizen of the transformation language. I discuss how the underlying transformation engine needs to be adapted in order to use the same language uniformly for different change scenarios.

The technicalities of the approach will be discussed on a (1) model synchronization case study and (2) a case study on detecting the violation of evolutionary (temporal) constraints. Both case studies stem from the security requirements engineering domain, and solve problems posed by Chapter 6.

This chapter follows [1].

7.1 Terminology of change in change-driven transformations

Changes are inherent to modeling. In model-driven engineering, models are rarely static, in fact, they are evolving continuously. Most of this evolution is driven by user input in modeling environments and editors. In other cases, changes are automatically introduced by batch model manipulations such as model import, transformation and export.

Change is considered to be the transition of a model from a pre-state, to a post-state, and the difference between the two is called the change delta (or model delta). This terminology is independent from the granularity and the abstraction level; it applies for changes that are just elementary model manipulation operations as well as for batch transactions encompassing a long sequence of transactions (see Section 3.1) or even for complex business decisions. In a long model manipulation sequence, any arbitrary intermediate stage can be considered the pre-state from the point of view of a change-driven transformation; but it is most useful to consider changes relative to the latest state when the
model was “consistent” for purposes of the transformation.

A model-driven design setup requires these changes to be propagated along a chain of tools into derived models or generated source code. The workflow may also involve the merging of models, back-annotating the results of an analysis performed on a transformation’s target model to the source model, or identifying interesting or erroneous parts within models. Thus there is a need for capturing the changes precisely.

In this chapter, a novel model transformation technology is presented that was designed to address this problem by operating on changes of models as first-class citizens. First, Section 7.1.1 proposes a classification scheme for changes that will be handled uniformly with change-driven transformations. This taxonomy will be useful to describe which cases the change-driven transformation approach aims to deal with, and what its advantages are. Building on this terminology, Section 7.1.2 explains the challenges of change-driven transformations.

### 7.1.1 Aspects of change

I define four perspectives (control, observability, information source, delta representation), distinguishing several different ways to perceive changes to a model. An overview is shown in Figure 7.1.

![Figure 7.1: Change scenarios (ignoring controllability)](image)

#### 7.1.1.1 The controllability perspective

There are scenarios where changes are controllable, meaning only an explicitly defined set of changes is permitted in each state of the model. A common example is when models are required to be edited exclusively using dedicated editors that only allow a limited set of high-level domain-specific model manipulation operations. Such modeling languages are often described by generating graph grammars [KM00, dLV02], where the grammar rules coincide with the editing rules.

However, in a wider range of scenarios, the transformation designer has no control over the possible ways a model may change during its lifecycle. It can happen through manual editing in a visual tool, batch refactoring, model transformation, model merging, etc. Any type of model manipulation is possible: creation/deletion of entities and relations of arbitrary type, modifying attribute values or element names, in any arbitrary sequence, in unforeseeable ways. Furthermore, it is even possible that models temporarily violate certain domain-specific well-formedness constraints during the changes. In this case, one needs to handle non-controllable changes.
7.1. TERMINOLOGY OF CHANGE IN CHANGE-DRIVEN TRANSFORMATIONS

7.1.1.2 The observability perspective

After the transformation is completed and any derived model(s) are created, it is possible that the source or target models are changed without any model management support (e.g., when the generated source code is changed in model-to-text scenarios). When the transformation is invoked next time, it can only access the current updated version (post-state), without having any additional information sources revealing how the models were changed since the last transformation execution. In this case, the change is invisible.

However, with support from a model management environment, there may be ways to trace the changes made to a model, such as change logs. When the transformation system has to determine the appropriate reactions to execute, it can take advantage of such information sources. The change is observable if it can be deduced what the pre-state was, what change delta has been applied to it, and what the resulting post-state is.

7.1.1.3 The source of information perspective

If the change is observable, further distinction is possible based on what kinds of information sources are available. As previously mentioned, a change consists of a pre-state, a post-state and a change delta between them. The change is observable if and only if at least two of these three information sources are directly available, since the third one can be derived. Although this derivation is possible, it might not always be efficient in an actual implementation. Therefore, three scenarios can be distinguished based on which two of these three information sources are provided by the modeling platform, acknowledging that each scenario offers a different kind of support for implementing change propagating transformations. A similar categorization is presented in [Men02].

Some model management systems may preserve a previous version of the model from the last execution of the transformation, in addition to the current version. This can be the case if version control is enabled in the model repository. When the pre-state and the post-state are directly observable, it is called the snapshot scenario (state-based in the terminology of [Men02]).

In other situations, a description of the change may be available before it was applied on the model. An example of such a situation would be applying a patch onto the model, that consists of changes performed on a remote copy of the model. This is also the case when change requests have to be analyzed in a change management system, before the changes are actually carried out. If the pre-state and the delta are directly available, it is called the command scenario (forward delta in the terminology of [Men02]), and the delta can also be called a change command.

Finally in the history scenario (called backward delta in [Men02]), the post-state is directly available along with the delta (which can be called a change history). A typical example would be manually editing a model in an editor environment, which produces notifications of the editing operations after they have been carried out, or saves transaction logs (e.g., redo stack) together with the updated version of the model.

It is a rare but possible case that all three information sources are directly available (this is called the change-based case in [Men02]). For example, an editor may save change logs, while the model repository captures the pre-state and the post-state as well. In this case, any of the implementation strategies proposed for the above three scenarios is applicable, and the choice can be made on the basis of efficiency.
7.1.1.4 The delta representation perspective

In the history and command scenarios, the change delta is available as an information source. In this case, we define a fourth perspective that indicates how the change delta is perceived by the model transformation environment.

In the documented change scenario, the delta is available as a data structure called the delta document, that specifies exactly how the pre-state and the post-state differs. One example (history scenario) is a model editor maintaining a redo log during editing, that may be retained when the model is saved. The previously mentioned change management system with change requests can be thought of an example in the command scenario.

In the live change scenario, the change is experienced on-the-fly, as it is happening, by a persistently existing transformation continuously receiving run-time notifications on the change. The notifications (e.g. method calls) can be issued before or after the actual change (command or history). The notification granularity (frequency) can range from the level of elementary model manipulations to aggregated effects of longer transactions, smoothly transitioning into the documented case. Such a live transformation can be found frequently in model editing environments and centralized model management solutions. As a great advantage of this scenario, changes to a source model can be on-the-fly reflected in the target model, and other kinds of live transformation can be performed efficiently, facilitating valuable feedback [15][HLR06, RÖV10].

7.1.2 Transformations of change

Change driven model transformations are model transformations which consume changes of the host model $M$ as input (see Fig. 7.2), and turn these changes into model manipulation operations, native operations (such as asynchronous messages, or external API calls), or traceability records for persistent storage of changes.

![Figure 7.2: Change-driven transformations](image)

Essentially, a change driven transformation rule is enabled by some changes in the host model. The actual change representation can be of different nature (in accordance with Figure 7.1), e.g. a sequence of model manipulation operations or a change delta.

7.1.2.1 Challenges for change-driven transformations

This interpretation of change-driven transformations needs to be refined in many practical application scenarios with different model handling characteristics, which are discussed in the following.
• **Unified handling of complex changes in all change processing scenarios.** Analogously to high-level formalisms of model and graph transformations, change-driven transformations should support a declarative, high-level specification of changes that can be seamlessly integrated into a "host" model or graph transformation language. Moreover, this formalism (and the underlying execution semantics) should support a uniform specification and execution model for all change processing scenarios discussed previously, in order to relieve the transformation developer from a significant amount of manual coding (notification, adapters etc.), especially in the case of non-controllable changes. As an additional benefit, this independence will make a transformation portable across different change scenarios without modifying its code.

The language can then be used as (i) a complete stand-alone formalism for handling model transformation scenarios such as incremental model synchronization, model simulation (animation) in discrete systems, and on-the-fly well-formedness constraint evaluation. Additionally, (ii) it is also useful as an intermediate formalism bridging the gap between the technical challenges of the different change scenarios and high-level languages tailored for certain uses of model transformations (e.g. QVT Relations for model synchronization, or other GT-based languages for behavioral simulation).

• **Ability to handle traditional model transformation scenarios.**

Ideally, the change-driven rule formalism should support traditional execution semantics as well (based on an empty pre-state), so that the rules can be used without additional changes e.g. to perform the "first" transformation phase in model synchronization scenarios.

• **Handling both materialized and non-materialized models.**

A typical assumption of most model transformation approaches is that the host model $M$ is available as a materialized model in a common model store (e.g. as in-memory EMF models inside the MT framework). However, in some model transformation scenarios, this may not be technically feasible (e.g. for performance reasons – the model may be too large to fit in memory, or not trivial to import and convert). Practically, this means that only an external interface of its native environment is available for querying and manipulating $M$, but still using some model transformation approach is desirable to incrementally synchronize the model (e.g. for maintaining consistent views).

• **Traceability models** are used ubiquitously in many MT scenarios for correspondence mapping and also to preserve some information on the execution state of the transformation itself. Often, the only use of this information is to help the specification of target-incremental rules so that they only operate on changed parts of the model (e.g. incremental change propagation in model synchronization). In invisible change processing scenarios, the proposed change-driven transformation technology will automatically maintain a cache containing the (historical) information about the pre-state. As a result, traceability models (as well as rule preconditions) can be simplified significantly: they are only used for correspondence mapping between source and target models, but not for storing the past. For instance, old values of attributes would no longer be required to be stored as part of the traceability model to support attribute changes, which is typically the case for existing transformation technology.

• **Checking properties of the model evolution** can also be a challenging task for change-driven transformations. Here certain constraints can be evolutionary in the sense that they need to be evaluated over a sequence of model evolution steps and not over a single snapshot.
of the model. Traditional constraint languages (like OCL) can only handle these properties by encoding the trajectory as part of the models, which may blow up models significantly.

In addition to providing support for traditional traceability use-cases, change-driven transformations also allow the changes themselves to be represented as models (attached to the host model on which they are evaluated). Moreover, the model-based representation should be completely equivalent to the in-memory representation of live changes so that both the "documented" and "live" change processing scenarios can be handled uniformly.

7.2 Language for change-driven transformations

7.2.1 Requirements and motivation for change-driven rules

For many transformation engineers, declarative, rule-based techniques may offer an easy-to-understand way to specify model transformations. Consequently, I propose such a high-level change-driven rule formalism where transformation rules are augmented with a guard. The guard is evaluated in context of the changes that the graph model has undergone to determine whether the rule is an appropriate reaction to the change. In rule-based expert systems, this idea of change as a distinguished representation of information has been used for decades; for instance, in the well-known terminology of Event-Condition-Action (ECA) systems [DGG95], the guards of the presented formalism correspond to the notions of "triggering event" and the contextual condition of rules. As a complete adaptation of these techniques to model transformation technology (which is able to handle all relevant change processing scenarios using a unified, high-level formalism) does not yet exist to my best knowledge, I believe that such a language – architected as an extension to an existing graph transformation language – will serve practical applications well, in a number of application scenarios (e.g. model synchronization [RVV09], on-the-fly constraint validation [15] and model animation [RVV08]).

There are a number of requirements that such a language needs to fulfill:

• reactivity to be able to specify dynamic model changes as events that activate a rule

• conciseness to result in compact specifications for change-driven transformations

• high-level specification to be able to abstract from irrelevant details

• intuitiveness so that rules can be easily understood by those who are familiar with other model transformation languages

• expressiveness in order to be able to specify a large class of change-driven transformations using this language.

In this section, I will propose a language for change-driven transformations. This language will be based on the query language proposed in Section 5.2 (albeit relation variables will be allowed, as the context is no longer only EMF), and its model manipulation capabilities will be an extension of the ViATRA2 transformation language [VB07]. A quick overview of the new language concepts is presented in Figure 7.3, which will be gradually discussed in the sequel: Section 7.2.2 defines change patterns, while Section 7.2.3 specify change driven transformation rules on the foundations of change patterns.
7.2. LANGUAGE FOR CHANGE-DRIVEN TRANSFORMATIONS

7.2.2 Change patterns

The high-level rule guards (preconditions) for change-driven rules are defined in the form of change patterns. In addition to conventional graph patterns matched against the post-state, guards should also contain constructs for expressing the difference between the pre-state and post-state in the form of change queries. An appearance query indicates a graph pattern with a new match in the post-state, while the disappearance query indicates that a match of a given graph pattern is invalidated by the change. Introduced later in Section 7.2.2.1 as a useful syntactical sugar for expressing changes to value assignment relations (i.e. attribute value update), an attribute update query captures that an attribute changes from an old value to a new one, i.e. it detects if an old value of an attribute disappeared, or a new value appeared.

The benefit of using graph patterns instead of elementary changes as appearance/disappearance queries is that a change pattern will match regardless of the order of elementary model manipulations that ultimately satisfied the appearance / disappearance / attribute update queries. Thus, in case a match appearance was detected, it is irrelevant what the last operation was that completed the match of the appearance query, and so on. As a result, a single change pattern compactly captures a large set of different change sequences.

Definition 73 (Graph change pattern) Graph change patterns (CP) can be defined as a tuple $CP = \langle P, P^+, P^- \rangle$, where

- $P$ is the **main graph pattern**, which is permitted to be disjunctive, have NACs, etc.
- $P^+$ is a set of graph patterns $\{P_i = \langle V_i, C_i \rangle\}$ called **appearance queries**. Each appearance query $P_i$ with variables (pattern elements) $V_i$ and their constraints $C_i$ represents that a certain graph pattern appears due to the change. $P_i$ is allowed to share variables with $P$.
- $P^-$ is a set of graph patterns $\{P_j = \langle V_j, C_j \rangle\}$ called **disappearance queries**. Each disappearance query $P_j$ with variables (pattern elements) $V_j$ and their constraints $C_j$ represents that a certain graph pattern disappears due to the change. $P_j$ is allowed to share variables with $P$. 
• Appearance and disappearance queries altogether are called **change queries**.

• The set of common variables of a given change query and the main pattern is called its **interface**. \( I_i = V_i \cap V(P) \), \( I_j = V_j \cap V(P) \).

• The **pre-state pattern** \( P_{pre}(CP) = \bigcup_{P_i \in P^\ast} P_i \cup P \) summarizes disappearance queries and the main positive pattern, i.e. all patterns representing existence in the pre-state.

• The **post-state pattern** \( P_{post}(CP) = \bigcup_{P_i \in P^\ast} P_i \cup P \) summarizes appearance queries and the main positive pattern, i.e. all patterns representing existence in the post-state.

The match of change patterns (Figure 7.4) is defined against a pair of graph models \( G_{pre} \) and \( G_{post} \) over the same metamodel \( MM \), such that \( G_{post} \) is derived from \( G_{pre} \) by some (maybe only observable, but not controllable) model manipulation. Thus the sets of model entities (\( Ent_{pre} \) and \( Ent_{post} \)) and relations (\( Rel_{pre} \) and \( Rel_{post} \)), all subsets of the same universe \( U \), may intersect on elements that were preserved by the change from \( G_{pre} \) to \( G_{post} \). Furthermore, the metamodel and the data algebra is the same in both cases. Here \( G_{pre} \) and \( G_{post} \) represent the pre-state and post-state respectively, but their presence in the definition does not imply that the concept of change patterns is restricted to the snapshot scenario (see Section 7.1) – only to unify the semantic discussion.

**Definition 74 (Match of change pattern)** A match of the Change Pattern \( CP = \langle P, P^\ast_+, P^\ast_- \rangle \) in \( \langle G_{pre}, G_{post} \rangle \) over the same metamodel \( MM \) is the mapping \( m = \langle m_P, m^+_i, m^-_i \rangle : CP \to \langle G_{pre}, G_{post} \rangle \), where

• \( m_P : P \to G_{post} \) is a match of \( P \), in the post-state \( G_{post} \): \( G_{post}, m_P \models P \).

• For each \( P_i \in P^\ast_+ \), the set \( m^+_i \) contains a mapping \( m_i : P_i \to G_{post} \), such that
  - \( G_{post}, m_i \models P_i \), i.e. \( m_i \) a match of pattern \( P_i \) in graph \( G_{post} \).
  - \( m_i(v) = m_P(v) \) for interface variables \( v \in I_i \), i.e. \( m_i \) interfaces with the match of the main pattern, and

Figure 7.4: Change pattern concepts

![Change pattern concepts diagram](image)
7.2. LANGUAGE FOR CHANGE-DRIVEN TRANSFORMATIONS

- \(G_{\text{pre}}, m \not| P\), i.e. the same \(m_i\) is not a match in the pre-state.

- For each \(P_j \in P^*\), the set \(m^*_j\) contains a mapping \(m_j : P_j \rightarrow G_{\text{pre}}\), such that
  - \(G_{\text{pre}}, m_j | P_j\), i.e. \(m_j\) a match of pattern \(P_j\) in graph \(G_{\text{pre}}\).
  - \(m_j(v) = m_P(v)\) for interface variables \(v \in I_j\), i.e. \(m_j\) interfaces with the match of the main pattern, and
  - \(G_{\text{post}}, m_j \not| P_j\), i.e. the same \(m_j\) is not a match in the post-state.

**Definition 75 (Pre-state and post-state match)** For a match \(m = \langle m_P, m^*_+, m^*_- \rangle : CP \rightarrow (G_{\text{pre}}, G_{\text{post}})\),

- the **pre-state match** is defined as \(m_{\text{pre}} = \bigcup_{P_j \in P^+} m_j\), i.e. the unification of the match components corresponding to the pre-state pattern \(P_{\text{pre}}(CP)\); consequently \(m_{\text{pre}}\) is a match of the pre-state pattern in \(G_{\text{pre}}\), i.e. \(G_{\text{pre}}, m_{\text{pre}} \models P_{\text{pre}}(CP)\);

- the **post-state match** is defined as \(m_{\text{post}} = \bigcup_{P_i \in P^*_{\text{+}}} m_i \cup m_P\), i.e. the unification of the match components corresponding to the post-state pattern \(P_{\text{post}}(CP)\); consequently \(m_{\text{post}}\) is a match of the post-state pattern in \(G_{\text{post}}\), i.e. \(G_{\text{post}}, m_{\text{post}} \models P_{\text{post}}(CP)\).

Note that this definition is deliberately asymmetric for \(G_{\text{pre}}\) and \(G_{\text{post}}\), as the main pattern \(P\) is interpreted on \(G_{\text{post}}\) only.

The concept of **matchable** attributed graph patterns (see Definition 31 on page 29) can be naturally extended to change patterns to restrict uses of attribute variables to cases that a pattern matcher can handle.

**Example 43** The following example is part of the case study of Section 7.3 about the bidirectional change propagating transformation (see Section 6.4) between a Si* and a ScCMER model. Figure 7.5(a) and Figure 7.5(b) show two CPs that detect newly created Si* Goals (to be mapped to a Goal entity in the ScCMER model), and deleted ScCMER Goals (in order to delete the corresponding Si* Goal), respectively. NACs are here visually represented as special sub-patterns (enclosed in a "NEG" box), while the rectangles marked by **appear** or **disappear** indicate that the enclosed subpattern is an appearance or disappearance query, respectively. As said earlier, the CP of Figure 7.5(a) is insensitive to the last operation that caused the Si* Goal to appear, be it the creation of the Goal entity, reassigning to a different Actor, etc. Listing 7.1 displays the same CP as Figure 7.5(a) with a textual syntax. As an extension to the graph pattern language of EMF-IncQuery (see Section 5.2), change queries are available as a (sub)pattern definition with **appear** or **disappear** prefixes, in a syntax analogous to NACs.

The change patterns presented here are somewhat simplified compared to what is actually required for the transformation problem. For instance, if an Si* Actor has a Goal that was delegated to it, then there is no corresponding **wants** relation in ScCMER, therefore a NAC should have been added to the appearance query of Figure 7.5(a).

**Example 44** For demonstration purposes, these CPs are matched against the transaction of Scenario 1 of Section 6.1.3.1, between the pre-state in Figure 6.1(a) and the post-state in Figure 6.1(b). The pre- and post-states of the Si* diagram are shown in Figure 6.1(a) and Figure 6.1(b), respectively. While the diagrams show the visual syntax of Si* only, it is assumed that there is a corresponding ScCMER model that is synchronized with the Si* one in the pre-state. It is further assumed that the transaction...
(a) Detecting creation of Si* goal

(b) Detecting deletion of S/e.scCMER goal

Figure 7.5: Example change patterns

(e.g. performed by the Si* graphical editor) changed the Si* model only, while the S/e.scCMER model has been unaffected; it is the job of a change-driven transformation to propagate the changes from the concrete syntax to the abstract model.

The CP in Figure 7.5(b) will not match against this change, as it contains a disappearance query in the S/e.scCMER domain, but the S/e.scCMER model was not changed. However, the CP in Figure 7.5(a) has a match. The post-state contains an Actor called “AMAN” in both models. A goal called “MDAccessControl” was added during the transaction in Si*, satisfying the appearance query. As the model manipulation did not change the S/e.scCMER part, the new Goal “MDAccessControl” is not yet part of it, so this constitutes a match of the CP. More precisely, the appearing variable \( g \) in the change pattern will be substituted for the Si* Goal \( MDAccessControl \), \( a \) for the Si* Actor \( AMAN \), \( A \) for its S/e.scCMER counterpart, while \( gN \) will be the string “MDAccessControl” and \( aN \) will be the string “AMAN”. The occurrence of the CP will trigger a transformation rule that will be responsible for creating a new Goal in the S/e.scCMER model.

Note that the post-state by itself is not enough to determine which of the two CPs have a match; in both cases, there is a Goal in Si* with no corresponding element in S/e.scCMER. If the CP in Figure 7.5(b) had a match instead, the correct reaction would be the deletion the Si* Goal as opposed to the creation of a S/e.scCMER goal. This demonstrates the added value of change patterns over regular graph patterns.
7.2. LANGUAGE FOR CHANGE-DRIVEN TRANSFORMATIONS

(a) Post-state pattern of Figure 7.5(a)

(b) Post-state pattern of Figure 7.5(b)

Figure 7.6: Post-state patterns of example change patterns

See also the similarity in the post-state patterns of the two CPs in Figure 7.6. The only difference between the post-state patterns is that Figure 7.6(a) has a NAC with the S/e.scCMER Goal, while the other post-state graph pattern could also possibly match if the S/e.scCMER Goal was present. Which is to be expected, since its associated CP does not exclude the existence of such a Goal, merely asserts that one such Goal has disappeared (there could potentially be others left, except for the fact that names are assumed unique in S/e.scCMER).

7.2.2.1 Extensions

Although not presented in the formal definition to provide better focus on the core contribution, there is a wide range of straightforward extensions to the presented version of the CP formalism (some of which will be used in subsequent examples). Similarly to graph patterns, it is possible to define negation, composition, disjunction, etc. for change patterns as well, but the formalization is omitted here for brevity. Without including a proof, it is worth pointing out that if these features are available, then the expressiveness of CPs becomes equivalent to first-order formulae over the set of predicates describing the pre-state and the post-state; in analogy to the first-order expressive power [Ren04b] of graph patterns over a single model.
CHAPTER 7. QUERIES FOR CHANGE-DRIVEN TRANSFORMATIONS

change pattern newSistarGoal(a,aN,A,g,gN) = {
  sistar.Actor.name(a,aN);
  secmer.Actor.name(A,aN);
  appear {
    find sistarActorHasGoal(a,g);
    sistar.Goal.name(g,gN);
  }
  neg {
    secmer.Actor.wants.name(A,gN);
  }
}

Listing 7.1: Textual version of the example change pattern of Figure 7.5(a)

This suggests that the CP formalism is powerful enough justifying the choice to be used to trigger change-driven rules.

A different family of extension does not extend the expressiveness of the language, but improves conciseness. One can imagine several kinds of composite change queries that are expressible with appearance and disappearance queries (elementary change queries) and graph pattern elements, yet are added to the language as useful syntactic sugar to enable more concise specification of change patterns. Attribute update queries are one kind of such composite change queries that are specifically aimed at expressing attribute updates:

Definition 76 (Attribute update queries) A change pattern specification can rely on attribute update queries $U^*_h$ that is a set of tuples $\{U_h = \langle v_{Mod}^h, attr_h, v_{pre}^h, v_{post}^h \rangle \}$. Each attribute update query represents that a certain model element has one of its attributes changed, where $v_{Mod}^h \in V(P)$ is a structural entity variable of $P$ that represents the model element, $attr_h$ is the attribute name, and the (optional) variables $v_{pre}^h, v_{post}^h \in V(P)$ represent the pre-state and post-state values of the attribute, respectively. The interface is $I_h = \{v_{Mod}^h, v_{pre}^h, v_{post}^h \}$. The attribute update query is equivalent to a single-constraint disappearance query of a value assignment relation of type $attr_h$ from $v_{pre}^h$ to $v_{pre}^h$, and a similar appearance query where the value assignment target is $v_{post}^h$.

Further composite change queries may include, for instance, a query for a pattern match that was present in both pre-state and post-state. This "persistence" change query is expressible by elementary change queries: it should be added to the main pattern $P$ and also as a negated appearance constraint, taking advantage of CP negation.

Similarly, statements can be made of what is present in the pre-state (regardless whether it still holds in the post-state).

7.2.3 Change-driven rules

Change-driven transformation rules are now introduced based on the formalism of change patterns. GT-style rules consisting of a CP as a LHS/guard (instead of a conventional LHS pattern) and a graph pattern as RHS are Change-driven GT rules (CDR). A CDR specifies a reaction to the CP used as its guard. As explained on Figure 7.7, the reaction is a controlled change transforming $G_{post}$ into an even newer state $G_{new}$. The transformation substitutes a match $m$ of the guard (more precisely, the post-state match $m_{post}$ of the post-state pattern $P_{post}(CP)$) with the image of the RHS pattern, using the same semantics as a GT rule application. In fact, the application of the CDR will be formally defined by a reduction to an application of a GT rule.
7.2. LANGUAGE FOR CHANGE-DRIVEN TRANSFORMATIONS

Definition 77 (Change-driven graph transformation rule) Change-driven graph transformation rules \( CDR = (CP, RHS) \) are specified by a guard change pattern \( CP = (PN, P^+_*, P^-*) \) defining the applicability of the rule, and a postcondition (or right-hand side) positive pattern \( RHS \) which declaratively specifies the result model after rule application. The post-state pattern \( P_{post}(CP) \) and \( RHS \) are allowed to share variables.

Obviously, the postcondition may only use/delete elements that are not already deleted when the guard matches, hence the usage of the post-state pattern. \( P_{post} \) and its match \( m_{post} \) in \( G_{post} \) will also be used to define the application of the rule. CDR application is the replacement of the post-state pattern with the RHS, or equivalently, the application of a conventional GT rule obtained from the change-driven rule with \( P_{post} \) substituted for LHS.

Definition 78 (Post-state reduction of a change-driven graph transformation rule) The post-state reduction of a change-driven graph transformation rule \( CDR = (CP, RHS) \) is the (conventional) graph transformation rule \( R_{CDR} = (P_{post}(CP), RHS) \), whose left-hand-side is the post-state pattern of the guard change pattern \( CP \), and the right-hand-side is shared with \( CDR \).

Definition 79 (Application of change-driven graph transformation rule) A change-driven rule \( CDR = (CP, RHS) \) can be applied on a guard match \( m = (m_P, m^+_*, m^-*) : CP \rightarrow (G_{pre}, G_{post}) \) after a change from pre-state \( G_{pre} \) to post-state \( G_{post} \). The application of the CDR results in a new graph model \( G_{new} \) derived from \( G_{post} \), where the transition from \( G_{post} \) to \( G_{new} \).
is identical to the application of the post-state reduction GT rule $R_{CDR} = \langle P_{post}(CP), RHS \rangle$ on post-state match $m_{post}$. If $CP$ has no matches in $\langle G_{pre}, G_{post} \rangle$, then $CDR$ is not applicable.

After applying a change-driven rule (or any other form of model manipulation), the current state of the model will be $G_{new}$, so it will play the role of post-state in further evaluations of change patterns.

**Example 45** Figure 7.8 and Listing 7.1 show the CDR that propagates transitions deleted in the workflow models to the jPDL domain. The guard CP, identical to Figure 7.5(a), activates whenever a new Goal appears in $S\text{I}^*$, which is still unmapped to a S\text{C}MER Goal. The RHS contains such a S\text{C}MER Goal, therefore it will be created when the rule is applied.

For example, as already discussed, the CP guard will have a match on the pair of states depicted in Figure 6.1(a) and Figure 6.1(b). The rule will be applied as a reaction, resulting in the creation of a S\text{C}MER Goal with the name “MDAccessControl” that is connected by a wants relation from the Actor “AMAN”. As a result, that the modification of the $S\text{I}^*$ model is successfully propagated to the S\text{C}MER model.

![Change-driven rule to propagate creation of a Goal in $S\text{I}^*$](image.png)
7.2. LANGUAGE FOR CHANGE-DRIVEN TRANSFORMATIONS

7.2.3.1 Extensions

Recalling that graph transformations in Section 2.2.4 were introduced merely as a special case of model manipulation operations (see Section 2.1.5), there is a natural extension of the change-driven graph transformation rule family towards a more general form of change-driven rules. For a given CP, any model manipulation operation that is parameterized by post-state matches of the CP can be considered a CDR with the CP as guard/precondition.

From the point of view of the actual textual language, the same issue can be raised. While the declarative specification of GT rules and CDRs can be very concise in some cases (especially with pattern reuse by composition), in other applications it is more practical to also associate imperative actions to the rule that should be executed on the match of the guard. Examples include logging or debugging, chaining related rules, performing nontrivial computation, etc. Therefore the transformation language used in the examples contains an extension to the core CDR formalism, so that an action sequence can be attached to the rules using the action keyword. This technique provides a complete imperative alternative to using the declarative RHS formalism.

Change-driven rules vs. GT rules. It is worth pointing out that both traditional GT rules and an earlier event-driven rule formalism (graph triggers in [15]) can be thought of as special cases of the more expressive CDR formalism. CDR rules reduce to GT rules in case there are no change queries, while graph triggers are equivalent to CDR rules consisting of an empty main pattern and a single change query (graph triggers use the appearance/disappearance of the entire precondition pattern as guard condition).

7.2.4 Challenges addressed

In the following, I summarize my arguments supporting that this transformation language extension answers the challenges of Section 7.1.2.1 and satisfies the requirements given in Section 7.2.1:

- **reactivity**: the transformation can react to changes in the model using change patterns as guards for transformation rules.

- **conciseness**: change queries capture the relevant information in the delta without the need for individually addressing possible sequences of elementary changes that have lead to the given post-state.
• **high-level specification**: model changes can be abstractly captured as appearance and disappearance of graph pattern matches; and change-driven transformations are also independent from the source of the triggering changes.

• **intuitiveness**: the proposed language extends declarative static model queries and model manipulation (as provided by graph patterns and graph transformation rules) in a natural way by introducing change patterns (which are guards that specify elements which must appear or disappear) and change-driven rules (which describe reaction to changes).

• **expressiveness**: change patterns allows the transformation designer to specify rules which can distinguish between identical post-states of the model based on the modification trajectories which led to that state, without (i) having to encode these modifications into complex traceability models and (ii) bloating transformation rules with them. In other words, change-driven rules extend the expressive power of graph transformation rules by high-level queries corresponding to the changes exhibited by the graph. (These queries have the full expressive power of first-order predicates over the pre-state and post-state model.)

### 7.3 Case study: bidirectional synchronization

The first motivating scenario is from the the ScCMER tool (see Chapter 6). Here extracts will be given from a change-driven live transformation that solves the bidirectional model synchronization problem (see Section 6.4 for the special challenges of this transformation) between the abstract syntax of ScCMER requirement models and the visual concrete syntax (Si* diagrams). Note that the paper [1] contains a different model-to-model transformation case study that demonstrates a different application scenario involving non-materialized models.

An important property of this change-driven transformation is that no correspondence or traceability model is required that maps Si* elements to ScCMER. This is a consequence of the fact that the correspondence can be established based on matching element names in the two models. CDRs can take advantage of this and eliminate the correspondence models. As the post-state of models does not always convey enough information to determine the correct action (see Example 44 for a demonstration), many other M2M formalisms may still require a correspondence model essentially to "remember the past", i.e. use it as a shadow copy in the terminology of Section 7.5.1.3. The following partial case study solution will show how CDRs achieve the same goal without maintaining and using a correspondence model, which would impose an overhead on transformation specification size (unless implicit, as with QVT), and also on model storage size and execution time. Note that the implementation strategy for CDRs that will be proposed in Section 7.5 would also induce such shadow copies, but only in the invisible scenario. Although in such a case there is no advantage over other approaches in the execution time and model size, but (a) this service is transparent and does not complicate the rule specifications, (b) the ScCMER tool, which serves as the current case study, would execute the CDT in the live scenario anyway, in which creating such shadow copies is not necessary.

The complete solution will not be detailed here, but change-driven transformation rules (and helper graph patterns and change patterns) will be specified that provide the following functionality:

- If an Actor is created in the ScCMER model, a corresponding Actor is created in Si* (specifically a Stakeholder by default, as an Actor could also be an Agent).

- If an Actor is deleted in the ScCMER model, each of the corresponding Actor elements (that all share the same name) are deleted in Si*.
7.3. CASE STUDY: BIDIRECTIONAL SYNCHRONIZATION

- If an Actor is renamed in the SeCMER model, each corresponding Actor element is renamed in the same way in Si*.

- If an Actor element (either Stakeholder or Agent) is created in the Si* model whose name was unused before, a corresponding Actor is created in the SeCMER model.

- If the last of the Actor elements having a given name is deleted in Si*, the corresponding Actor element is deleted in the SeCMER model.

- If the Actor elements in Si* sharing a given name are renamed together (the Si* editor does this as a single transaction), the corresponding Actor in the SeCMER model is renamed along with them.

Only high-level specifications will be given for the action parts of CDRs; they will not be detailed here, as they contain miscellaneous technical tasks that are not important from the point of view of the thesis. Nevertheless, it should be noted that (a) in case of deletions, all connections/references to the deleted element should be properly removed, (b) the manipulation of the Si* model must happen through Si* API calls rather than by direct model modifications, as the visual appearance of the diagram has to be maintained accordingly.

The first of these rules is shown in Listing 7.2. The helper pattern `sistarActorName` matches any string that is the name of a Si* actor; note that for any given string, there will be up to one match of this pattern regardless how many duplicate Si* Actors share that name. If a new match of this pattern appears, that means that there is a new Si* Actor that is not simply a duplicate of existing ones, and thus must be propagated to SeCMER. The action part (not shown) of CDR `newActorFromSistar` does just this: creates a new SeCMER Actor in the SeCMER model and sets its name to the value of variable `name`. Note that the guard CP has an additional NAC constraint to make sure that such a SeCMER Actor does not exist yet; this constraint is important to add since it is possible that a single transaction modifies the SeCMER model in synch with the Si* one. This can occur if e.g. a new part of the requirement model is loaded from a file where both representations are already available, or simply if the same rule has already been applied earlier in the transaction (see Section 7.5.3.2).

```plaintext
1 pattern sistarActorName(name) = {
  sistar.Actor.name(_a,name);
}

2 cdrule newActorFromSistar = {
  guard change pattern newSistarActor(name) = {
    appear find sistarActorName(name);
    neg secmer.Actor.name(_anyActor,name);
  }

  action {
    // create new SeCMER Actor with given name
  }
}
```

Listing 7.2: Propagating creation of Si* Actor

When one of the Si* Actors is deleted, while others with the same name remain, the corresponding SeCMER Actor should be preserved. The pattern `sistarActorName` will only lose a match when the last duplicate copy of an Actor is removed from Si*. This triggers the CDR in Listing 7.2, provided that the corresponding SeCMER Actor exists, to delete it as well.
CHAPTER 7. QUERIES FOR CHANGE-DRIVEN TRANSFORMATIONS

Listing 7.3: Propagating deletion of Si* Actor

Similarly to the above, the CDR in Listing 7.4 will be triggered if a SeCMER Actor is removed with one or more corresponding Si* Actors still present; the latter will be deleted by the action part. As discussed earlier, here the action part must interface with the Si* API to make sure that the deletions are displayed correctly by the diagram.

Listing 7.4: Propagating deletion of SeCMER Actor

When a new SeCMER Actor appears without a corresponding Si* Actor, the CDR in Listing 7.5 will propagate this change. Note that creating only one Si* Actor with the given name or having e.g. 58 of them would be equally valid, as both versions of the Si* model would be compatible with the same SeCMER model, without any changes to be propagated according to the transformation requirements in Section 6.4. The fact that the rule creates a single Si* Actor only is a choice of the transformation developer. Moreover, there are two kinds of actors in Si*, Agents and Stakeholders. These are not distinguished by the SeCMER model, a SeCMER Actor is compatible with either kind in Si*; the fact that the rule creates a Stakeholder by default (which can later be switched to an Agent if required) is once again a choice of the transformation engineer.

Listing 7.5: Propagating creation of SeCMER Actor

The CDR of Listing 7.6 propagates an attribute change, using attribute update queries and persis-
tence queries (both composite change queries are introduced in Section 7.2.2.1). If the SeCMER Actor is renamed, while the Si* Actor keeps its original name, then the renaming should be propagated once per each corresponding Si* Actor. Although name is used as the primary identifier of these Actor entities, and thus renaming one of them breaks the correspondence, CDRs can propagate this kind of change as well thanks to the expressiveness of CPs. It is important to point out that whenever this rule is applicable, the previously introduced rules delActorToSistar and newActorToSistar would also be applicable. They would delete the Si* Actors and recreate them under the new name. For purposes of target incrementality, the preferred action is to simply rename each corresponding Si* Actor; this preference is expressed by raising the priority (see Section 7.5.3.2) of this rule.

```plaintext
ecrule renameActorToSistar = {
    guard change pattern renamedSeCMERActor(secmerActor, sistarActor, oldName, newName) = {
        secmer.Actor(secmerActor);
        update secmerActor.name from oldName to newName;
        persist sistar.Actor.name(sistarActor, oldName);
    }
    action {
        sistarActor.setName(newName);
    }
    priority +10;
}
```

Listing 7.6: Propagating SeCMER Actor renaming

Finally, Listing 7.7 propagates name changes in the reverse direction. The rule assumes that all Si* Actors that represent the same SeCMER Actor are renamed atomically together in a single transaction (otherwise their equivalence would be broken). Fortunately, the renaming functionality in the Si* diagram editor observes this assumption.

```plaintext
drule renameActorFromSistar = {
    guard change pattern renamedSistarActor(secmerActor, sistarActor, oldName, newName) = {
        sistar.Actor(sistarActor);
        update sistarActor.name from oldName to newName;
        persist secmer.Actor.name(secmerActor, oldName);
    }
    action {
        secmerActor.setName(newName);
    }
    priority +10;
}
```

Listing 7.7: Propagating Si* Actor renaming

### 7.4 Case study: change impact analysis by evolutionary constraints

A second case study stems also from the SeCMER tool (see Chapter 6). An important role of security requirement models is to support reasoning about security properties by argumentation techniques in an early stage of development. The case study will focus on analyzing the impact of requirement changes on these argument models. Here a solution will be provided for the evolutionary constraint introduced in Section 6.3.
The challenge is to provide a straightforward and efficiently evaluated declarative language for this purpose. I propose on-the-fly, incremental evaluation for a wide range of evolutionary constraints that can be implemented as an efficient reactive mechanism by using change patterns and change-driven rules.

The proposed solution to the problem of Section 6.3 will be described in the following. A set of CDRs will be established to flag invalidated arguments as invalid and request argumentation analysis. The key difference between each of these rules is the change pattern used as the guard. The recommended strategy is to identify types of changes that guarantee a re-evaluation of the argument, and define a rule for each of them.

To aid in building CPs and CDRs, some helper graph patterns are defined first. GP `validArgument(A)` captures an `Argument A` that has not been invalidated (using metamodel elements that were omitted here for brevity). GP `evidenceOfArgument(A,E)` captures an argument `A` and a model element `E` which it references as an evidence.

The CDR `invalidateUponEvidenceUpdate()` is activated when an attribute of an evidence element is updated. The rule is guarded by a CP that contains an attribute update query linked to a match of `evidenceOfArgument(A,E)` and `validArgument(A)`. Listing 7.8 shows an initial version of this rule in a simplified syntax.

As an example for the application of this CDR, suppose that the evolution described in Figure 6.1.3.3 is carried out. This means that `Argument ARG` is a match of `validArgument(A)`: at the same time, `ARG` and `T_1` ("Perform a quarterly security screening of employees") constitute a match of `evidenceOfArgument(A,E)`. Whenever an evolution updates any attribute of this Task (e.g. downscale to yearly screening to cut costs, as in the example), the attribute update query will detect this, making `(ARG, T_1)` a match of the change pattern `evidenceUpdated(A,E)` and activating the CDR. The rule will flag the argument `ARG` for re-evaluation; argumentation experts will be alerted to revisit the argumentation and decide whether the looser policy is enough to maintain security needs.

Likewise, the CDR `invalidateUponEvidenceDeletion()` is activated when an evidence element is deleted. The rule is guarded by a CP that contains a disappearance query of `evidenceOfArgument(A,E)` linked to a match of `validArgument(A)`. Listing 7.9 shows an initial version of this second rule in simplified syntax.

Due to the flexibility of the CP formalism, additional similar rules can be created depending on system-specific policies; for instance the argument should be invalidated if an evidence element is
7.5 Implementation strategies for evaluating graph change patterns

The following sections outline a system architecture that implements change-driven transformation. Solutions to the following task items are required:

1. (positive and negative) graph pattern matching of the CP’s main pattern in the post-state
2. evaluating and matching appearance and disappearance queries (and composite change queries such as update queries)
3. matching change patterns, relying on the solutions to the above two tasks
4. applying change driven rules on matches of the guard change pattern

The first two of these tasks require different implementation techniques in different change scenarios (see Section 7.1), to take advantage of the benefits and avoid unnecessary operations that may degrade performance. First, Section 7.5.1 discusses the proposed solutions to the first two tasks in all change scenarios except for the live case. Next, Section 7.5.2 still addresses the first two tasks, but focuses on the live scenario with its unique execution model. Finally, the last two task items are addressed in Section 7.5.3.

7.5.1 Change query evaluation in documented or invisible change scenarios

7.5.1.1 Documented history and command scenarios

In the documented change scenarios, either the pre-state or the post-state of the model is available along with a delta document recording the changes. In order to match appearance and disappearance queries, existing graph pattern matcher algorithms have to be slightly modified.

The modified matcher algorithm have to consider the elements of the available snapshot and also the elements that only occur in the delta document, essentially unifying the pre-state and post-state.
Elements of this unified model can be classified as unchanged, deleted or created. In case of documented history, elements of the post-state that were created in the change history are classified as created elements, elements of the post-state that were not created in the delta are considered unchanged, and elements only appearing in the delta are classified as deleted elements. In the change command scenario, elements in the pre-state that are deleted by the command are classified as deleted elements, elements in the pre-state that are not deleted by the command are unchanged, and elements only present in the delta are classified as created elements.

A match of a positive graph pattern is only considered valid in the post-state, iff it contains no deleted elements. A pattern with NACs has a valid match in the post-state iff it is a valid post-state match of the positive pattern, and all NAC matches (if any) are disappearing (see later). A pattern match is considered appearing iff it is valid in the post-state (as defined above), and contains at least one created element, or has at least one NAC match (which is – as stated above - disappearing). Finally, a pattern match is considered disappearing if it contains no created elements, all of its NAC matches (if any) are appearing, and there is either at least one deleted element of the positive match, or a NAC match (which must be appearing). Other kinds of pattern composition (including aggregation) can be treated analogously.

Using these rules, the pattern matcher can determine the match set of the main pattern in the post-state, as well as that of the appearance and disappearance queries. The evaluation of composite change queries can be derived from elementary change queries; although in some cases, more efficient procedures can be given.

7.5.1.2 Snapshot scenario

In the snapshot scenario, both the pre-state and the post-state are directly available, therefore change query evaluation is reduced to fairly simple steps. An appearance query is satisfied if the pattern is matched in the post-state, but the same match is invalid in the pre-state; and vice versa for disappearance queries. Therefore evaluating these queries require pattern matcher techniques similar to matching NACs in regular graph patterns.

However, one of our assumptions here was that if a model element exists in both states, it is trivial to recognize that they are in fact the same element; definitions in Section 7.2.2 assumed that they are in fact the same element of the universe $U$. In some technological contexts, this assumption may pose a challenge, but adherence is possible if model elements have a unique identifier that is preserved across different versions. Unfortunately, in some modeling environments this is not guaranteed; for example, generic EMF objects are not identifiable by default in a way that is valid across snapshots (but fortunately EMF provides both live notifications and redo stacks instead). In this case, the two versions of the model have to be reconciled against each other (by either a generic heuristic or a domain-specific way) before the changes can be computed and CDR can be applied; see the related literature on model comparison [ASW09].

7.5.1.3 Invisible change scenario

As only the post-state is available, post-state matching of the main pattern is trivial in this case, but evaluating change queries is not. The common solution to this problem has significant time and space overhead: the transformation creates a shadow copy of the model each time it is invoked. On the next transformation run, the model itself represents the post-state, but the shadow copy preserves the pre-state, therefore the change queries can be evaluated. Of course, there is no need to replicate the entire model; it is sufficient to store the match sets of patterns used as appearance and disappearance
queries. The appearance and disappearance queries can be evaluated by matching the patterns against the post-state and comparing the match set against the one preserved in the shadow copy. Likewise, update queries can be evaluated by comparing the current attribute value against its shadow copy.

To prevent inconsistencies, the shadow copy should be inaccessible to normal model editing operations, which can be achieved either storing it separately (e.g. in a different file), or by using special model element types, markers, etc., that visual editors and other transformations ignore. If it is stored separately, the problem of preserving model element identity has to be dealt with, similarly to the snapshot scenario.

A widespread practice [Tra08, ELF08] in model-to-model transformations is to use the traceability model (sometimes called reference model or correspondence model) in a way that it preserves the LHS (or a significant subset thereof) of all executed rules. Thus the traceability connections essentially store a copy of the source model (or at least relevant match sets), thereby providing a shadow copy functionality. In those model transformation approaches where this is not handled automatically, significant manual effort is required for maintaining this shadow copy. With change-driven transformations, however, the platform can provide change queries as a service, hiding implementation details. The hidden implementation will involve an automatic shadow copy mechanism in the invisible change scenario (and less resource intensive solutions in the other change scenarios). On the rule specification level, the availability of change queries may allow a much simpler maintenance of traceability in many cases (especially bidirectional synchronization), sometimes as simple as using the same name for a source and target element, as there is no need to manually preserve the entire LHS.

### 7.5.2 Change query evaluation in live change scenarios

#### 7.5.2.1 Challenge of live scenarios

While all techniques for the documented scenarios are functionally correct in the live scenario as well, there may be an additional important requirement in this case. Live notifications can be used to perform live transformation, meaning that change-driven rules are executed on-the-fly. Since live notification is received about changes that are in progress, and reactions are triggered during an interactive session, pattern matching is required to be responsive and efficient. We propose an architecture capable of efficiently matching change patterns and applying change-driven rules with live monitoring of the model as it evolves.

The entire architecture is illustrated in Figure 7.9. The rest of Section 7.5.2 discusses how change queries are evaluated efficiently in live scenarios.

#### 7.5.2.2 Incremental pattern matching

As discussed earlier and shown by benchmarks (see Section 3.5), incremental pattern matching can improve performance or scalability by several orders of magnitude in certain scenarios; with a continuously evolving model, its benefits are likely to outweigh its drawbacks. Moreover, incremental pattern matching leads to easy discovery of appearing or disappearing pattern matches, thus it can be used to efficiently implement the change query feature of change patterns by incremental calculation of match set deltas (e.g. the Rete algorithm discussed in Chapter 3 computes the match set deltas alongside with the match sets).

Having received change notifications, the incremental pattern matcher shows an up-to-date picture of the post-state. This is true even in the command scenario where these changes might not have been applied to the model itself yet, still retaining the pre-state. Therefore the post-state graph
pattern matching of the main pattern can be performed by the incremental pattern matcher in both history and command scenarios. In the history scenario, the model itself reflects the post-state, therefore an unmodified non-incremental (local search) pattern matcher is also applicable for this task, if it is preferrable for performance reasons.

7.5.2.3 Delta monitoring

A delta monitor is a component that can be attached to a match set cache of the incremental pattern matcher at any time, and it will start to feed on the match set deltas provided by the incremental matcher to cumulatively record changes affecting the match set from that time on. At any point in the future, the delta monitor will be able to report which new matches appeared and which matches disappeared since it was initialized. If delta monitors are initialized when the model is in its pre-state, it will be able to continuously provide the delta match set between the pre-state and the specific post-state that is reached after the latest live notifications.

The tasks of the delta monitor are efficiently achieved by hooking into the internal notification/update mechanism of the incremental pattern matcher. Changes of a single match (e.g. the same match appearing and then disappearing later) may invalidate each other, therefore the delta monitor really reflects (a projection of) the delta between the two states, and not just recorded history.

7.5.2.4 Change query evaluation

Change queries can be efficiently evaluated using delta monitors and incremental pattern matching. Before the change is performed or notifications are received, i.e. in the pre-state, a delta monitor is to be attached (or reinitialized, if already attached) onto the incrementally maintained match set of each graph pattern that occurs as a change query within a CP. After the change, the contents of the delta monitors will reflect the graph pattern matches that have appeared or disappeared. This complements
7.5. IMPLEMENTATION STRATEGIES FOR EVALUATING GRAPH CHANGE PATTERNS

the post-state reflected by the incremental pattern matcher (or alternatively, in the history scenario, the model itself), to provide all necessary information for matching change patterns.

7.5.3 Implementing change-driven rules

7.5.3.1 Matching guard change patterns

Change patterns are equivalent to an extended graph pattern formalism, where the set of admissible pattern constraints contains change sets of pattern matches as special composition constraints. In the end, the match set of a change pattern can be determined from the match set of the change queries and the main positive pattern; and such a pattern matcher architecture is conceptually similar to existing ones dealing with negative application conditions or other pattern call constraints. Therefore graph pattern matching mechanisms can be used to evaluate change patterns, based on the partial solutions (change queries and post-state pattern matching of the main pattern) obtained differently in each change scenario.

7.5.3.2 Rule execution

The sequence of elementary model manipulation operations executed by any transformation unit, GUI-based manipulation, model merge or other job can be arbitrarily segmented into transactions, that are assumed to result in a consistent state of the affected model. The transaction is the unit of change that CDRs will react to; the starting and the end points of the transaction will be considered the pre-state and the post-state, respectively. In documented change scenarios, the whole change process between the given pre- and post-states can be considered a single transaction. In live scenarios, as notifications may be continuously sent, it is a nontrivial question how to segment transactions; it helps if there is some support for explicitly defining transaction boundaries and commit points. A typical transaction can be e.g. the execution of single functionality through the UI, corresponding to multiple elementary operations.

Upon the end of each transaction, the change patterns are evaluated to determine which change-driven rules are applicable. If there are any such CDRs, they are applied on the model, using algorithms that are identical to regular GT rule application. As this rule application phase modifies the model, there are two important execution schemata to distinguish regarding the transactionality of rule executions.

One strategy is to take one applicable CDR, and prolong the transaction by appending this rule application. If there are multiple applicable rules, the choice might be made according to some strategy, e.g according to rule priority. While the pre-state is unchanged, the post-state is advanced by the effects of the rule, thus the set of applicable rules may change. Once again, a rule application is selected; this loop can go on until there are no more applicable rules. At that point, the transaction can finally conclude; the newly reached state will be considered the pre-state henceforth. This execution strategy guarantees the important postcondition that there are no applicable rules at the boundaries of transactions. This can be used as a very useful precondition in defining the rules that will be applied in reaction to the next transaction.

According to an alternative approach, all applicable rules can be executed (effectively in parallel) after closing the original transaction; this rule application phase can be considered a change transaction itself, with its effects wrapped into a separate transaction. At the end of this second transaction, the effects of executed CDRs can be reacted to as well, as long as there are triggered CDRs.

In both cases, the rule application loop is actually a live scenario, regardless of the circumstances of the original triggering change. Execution schemata have been previously elaborated in [15].
7.6 Discussion

The potential advantages and limitations of change-driven transformation over traditional model transformation techniques will be discussed here.

7.6.1 Theoretical discussion

In this theoretical discussion, the primary focus is on graph transformation based approaches, which provide the closest correspondence to the new techniques, moreover, they have a sound, well-established underlying theory.

Theoretical expressiveness of change-driven graph transformations

While not formally proven here for space considerations, it is worth pointing out that for each change-driven graph transformation system (CDGTS) a corresponding graph transformation system (GTS) can be derived, which simulates the CDGTS. As a result, the proposed change-driven graph transformation rule formalism is not more expressive in a pure theoretical point of view, which is hardly surprising since GT rules are already Turing complete.

The construction relies on the concept of shadow copies: it essentially stores an explicit representation of the pre-state as a dedicated part of the model. Then a separate set of GT rules would be responsible for (1) the detection of change and management of pre-states and post-states (based on the difference between $G_{pre}$ and $G_{post}$) and (2) simulating the effect of a CDTS (based on the difference between $G_{post}$ and RHS). As a consequence, both the underlying model and the rule set would explode.

Note that the above construction required a change in both the metamodel and the instance models. Without this special encoding, the GT rules are less expressive as they take only the post-state into account to determine which action to take while CD rules can refer to the pre-state as well.

As a side remark, a model-to-model CDTS can be constructed even if the model transformation problem itself is non-deterministic by its nature (e.g. a tree-based hierarchy model needs to be flattened to an arbitrary sequence respecting the partial order induced by the hierarchy). Thus the correspondence relationship from source to target models is not necessarily one-to-one. The case study of Section 7.3 provides an example of such nonfunctional correspondence, with an arbitrary number of Actor duplicates in the $Si^*$ model.

Analysis of change-driven transformations

The main practical relevance of this simulation property from CDGTS and GTS is that it enables to investigate traditional semantic properties like termination or determinism using the rich theory and proof techniques (e.g. [VVGE+06, HKT02]) of graph transformation systems. For instance, if the simulating GTS can be proven to be terminating, then the original CDTS must be terminating as well; while GTS termination is undecidable in general, there are established proof techniques (see [VVGE+06]) for concrete systems. As a consequence, I believe that some existing analysis techniques of GTS are reusable for CDGTSs. Exploring this idea in detail is left as future work.

Constraint detection by change-driven transformations

Declarative specifications (like graph patterns, graph transformation rules, OCL constraints) are frequently used for detecting the violation of well-formedness constraints in domain-specific models.
However, evolutionary constraints related to the temporal behavior or the evolution of models are very hard to specify and detect as it requires to explicitly encode the sequence of model snapshots as part of the model, and thus also part of the constraint. Change patterns provide a direct and succinct way to detect a class of constraints related to the trajectory of model evolution (see the case study of Section 7.4).

**Relationship with temporal logics**

As an alternative to the language of change patterns, there is a wide family of temporal logics. They are able to express temporal relationships of more than two model snapshots. On the other hand, commonly used temporal logics can only describe a single model snapshot with propositional logic formulae, while graph patterns are equivalent to more powerful first-order logic formulae [Ren04b]. It is of course possible, though uncommon, to use temporal logics based on first-order formulae as well, thus they are strictly more expressive than the proposed language of graph change patterns.

The use case of change-driven transformations, however, does not require formulae over an arbitrarily long chain of model snapshots; a pre-state and a post-state are sufficient to express how the current state differs from a previous, consistent state. Therefore change-driven transformations would not take advantage of the increased temporal expressiveness. The language of change patterns was chosen over temporal logics for its simplicity, conciseness and the availability of efficient execution strategies in the various change scenarios.

**7.6.2 Expressiveness wrt. model synchronization languages**

As a single CDT rule is unidirectional, CDT specifications of bidirectional M2M transformations can be more complex than that of high-level bidirectional approaches (like TGG or QVT Relations), as two separate rule sets are required for change propagation in the two directions. The case study of Section 7.3 also demonstrated that separate rules are required for propagating creation, deletion and attribute update. In concise and high-level formalisms such mapping relationships may be expressible using a single rule.

However, CDT specifications are also not as verbose as they appear to be at first sight. First, the change pattern of a CDT rule compacts many different change trajectories, and only triggers for reaction once a complex (aggregated) change has been detected (disregarding the order of elementary changes). Consequently, traceability representation can be significantly simplified in contrast to TGGs and QVT (and often altogether omitted), since traceability models are not required to contain complex information about change trajectories (in contrast to e.g. [Tra08]).

Furthermore, different changes requiring the same reaction can be grouped together in one CDT rule (e.g. as an extension to the case study of Section 7.3, a single pair of slightly more general rules can be used to propagate the deletion of several element types in both directions).

TGG and QVT model synchronizations handle deletion of source elements by fully revoking the effects of the corresponding synchronization rules. As a result, the dependency between TGG rules has a significant effect on which parts of the target model need to be removed as a consequent undo action. CDTs allow a more fine-grained and explicit control for delete and move operations in source models to significantly reduce the amount of undos in the target model (by allowing temporal inconsistencies, for instance); see the case study of Section 7.3 for examples of such fine-tuning.

Finally, in real-world transformations, often very different reactions are required for each kind of change (creation, deletion, edge redirecting, etc.) and each direction of the propagation. This might be due to technological hurdles, e.g. properly interfacing with a diagram editor during live
transformation. An additional reason is to encode defaults and choices made by the transformation engineer in case the correspondence relationship between source and target models is not a one-to-one function. The case study of Section 7.3 provides examples for both cases. In very high-level formalisms, it is difficult to express such level of fine-tuning. Therefore TGG or QVT Relations might be used to derive an initial set of CDR rules, but CDR-level fine-tuning might be required afterwards.

Change patterns vs. elementary changes as guards.

A naive approach would be to use elementary change events (e.g. element creation / deletion) as guards [Egy06]. It is not a priori known which kind of elementary model manipulation operation will eventually trigger a transformation step. Therefore this low-level formalism forces us to define a separate copy of the transformation rule (or very complex disjunctive preconditions) for each possible triggering elementary change, and augment each rule with a check to see whether the elementary change really triggers the reaction. The high-level formalism of change patterns can trigger reaction when a compound event occurs, thus it significantly compacts the specification of guards.

Causality and dependency between CDT rules

First, causality and dependency between CDT rules can be handled implicitly using some traceability links in change patterns, which is conceptually similar to the TGG approach, and it does not require the additional use of when and where clauses as in QVT Relations. However, dependency and reusability are offered on the (change-)pattern level, thus complex main patterns, appearance and disappearance patterns can be assembled using pattern composition. Furthermore, CDT rules in the more general sense may (imperatively) call an arbitrarily complex batch transformation as a reaction to a specific aggregated change.

In the future, further investigation is needed in how existing MT languages can be translated into CDTs to further reduce the complexity of CDT specifications in case of model synchronization.

7.6.3 Practical discussion

From a practice-oriented viewpoint, the following discussion will investigate (1) the traceability representation between source and target models, (2) the representation of the pre-states for model synchronization scenarios.

The most apparent advantage of change driven transformations compared to traditional declarative model transformation approaches (like TGGs or QVT Relations) is that they impose significantly weaker assumptions on the nature of traceability models required during the transformation. Both TGGs and QVT require a real mapping (correspondence) model to interconnect source and target models with typed traceability links which need to be persisted either in a model store (in case of TGG) or in the transformation context (in case of QVT). Furthermore, both of these approaches to model synchronization are driven by the traceability information between the source and target models. For instance, if a source (or target) element is freshly created, it is detected by the lack of corresponding traceability element. Alternatively, the deletion of a source (resp. target) element can be observed by a dangling traceability element, which is only linked to a target (resp. source) element. This means that a large amount of information about the past is stored explicitly as part of the traceability model in the model store (or transformation context) in case of traditional model synchronization approaches.

In case of CDTs, the effects of transactions are propagated incrementally to the change patterns and change-driven rules, and instead of storing information about the past, it is the change in the match sets of patterns and rules which can be observed to trigger synchronization. Depending on the
actual change scenarios, CDTs significantly reduce what information needs to be stored about the past in model synchronization problem, since the traceability model is no longer used to “remember the past”. As a consequence, explicit traceability information can be reduced.

In most cases, traceability links for CDT can be as simple as pairs of source and target elements; these mapping can often be untyped, they can be stored in an external repository (independently of the model store or the transformation context), which may only persist the unique identifiers of source and target model elements.

As the extreme case, which is demonstrated by the case study in Section 7.3, there is no need for a traceability model at all. Traceability can also be provided on-the-fly by a function (e.g. a naming convention or identifier map) between the source and target models without persisting traceability information to a dedicated store. As a result, in case of CDTs, source and target models can be almost fully detached from each other in case of model synchronization scenarios using very simple traceability links or on-the-fly, non-persisted traceability information (traceability function) [HHRV12].

Note that in the invisible change scenario, the underlying change pattern evaluation mechanism will automatically manage shadow copies to “remember the past” like e.g. TGG traceability models (see discussion of shadow copies in Section 7.5.1.3). However, similarly to QVT, this is completely transparent, and has no impact on the transformation specification. If better guarantees can be made about the observability of the change, such as in the snapshot scenario (the pre-state of the model is preserved in version control) or in the live transformation scenario, no shadow copies will be stored.

7.7 Related work

Now an overview is given on various approaches showing similarity to the proposed change-driven transformations.

Event-driven techniques

Event-driven techniques, which are the technological basis of change-driven model transformations, have been used in many fields. In relational database management systems (RDBMS), even the concept of triggers [GMUW08] can be considered as simple operations whose execution is initiated by events. Later, event-condition-action (ECA) rules [DGG95] were introduced for active database systems as a generalization of triggers, and the same idea was adopted in rule engines [SB05] as well. However, ECA-based approaches lack the support for triggering by complex graph patterns, which is an essential scenario in model-driven development.

High-level transformation requirements and specification.

There are numerous high-level properties of bidirectional transformations that may be desirable in certain contexts. Such requirements include Undoability, Hippocraticness, etc. formulated in [Ste10]; see [Dis08] for an algebraic treatment of several properties and their relationship with transformation composition. The approach of change-driven transformations does not inherently guarantee any of these properties, as it expresses transformation rules on a lower level of abstraction. As a benefit, this offers more freedom; e.g. the change impact analysis case study of Section 7.4 could not have been implemented with restrictions like History Ignorance. On the other hand, if any of these properties are found desirable, then the burden of ensuring their satisfaction lies on the transformation developer.

Both event-driven transformation specification and manual enforcement of some of the previously mentioned high-level transformation requirements can be avoided by using very high-level transfor-
Inconsistency management

Inconsistency management systems aim at ensuring the consistency of multiple views of a software, which is designed by several engineers using tightly or loosely integrated tools. Views can be formulated on different levels of abstraction, and a bidirectional consistency of views is maintained by inconsistency detection and resolution.

Since these systems should typically support informal (e.g., natural language-based) descriptions as views, inconsistency resolution can never be fully automated, and manual user interaction in certain scenarios is unavoidably required, in contrast to the CDT approach, which automatically propagates and transforms change descriptions in a well-defined, rule-based way to the target domain to avoid the appearance of inconsistencies in the target model. Note that this is not necessarily an advantage of CDT, but rather a difference in focus.

[GHM98] presents a characteristic representative of inconsistency management systems, which records modification histories in the form of (model-based) change description objects (documented change in the current terminology). In contrast to CDTs, [GHM98] additionally saves and stores the detected inconsistencies for their possible resolution at a later time. The so-called grouping of inconsistencies in this approach would possibly allow for reaching a goal that is similar to the aim of pattern matching in the current chapter, however, in [GHM98] grouping is only used for presentation purposes, i.e., to create change and inconsistency lists for users to interact with.

[OG02] provides a conceptual architecture and prototype for supporting traceability and inconsistency management between software requirements descriptions, UML-style use case models and black-box test plans. Relationships between high-level software artefacts are represented by traceability links, which can be defined manually or in a semi-automated way. In contrast to the CDT solution, this approach supports change notifications on a low abstraction level, it can transform only simple modifications automatically, while other changes still need developer intervention.

In graph transformation, [GdL07] presents an approach for consistency management between abstract and concrete syntax representations of visual modeling languages. By their approach, the commands executed through the user interface are explicitly materialized as special command model elements and then processed by triple graph grammar (TGG) rules. This approach is a prime example for the "controlled" change processing scenario, where all possible editing operations are a-priori known; in contrast, the CDT technique primarily targets non-controllable change processing.

[ELF08] deals with consistency maintenance in UML models. This paper proposes target incremental techniques to efficiently detect inconsistencies and derive proposed corrections; recommended changes are represented explicitly (such as "DoesExist" and "ShouldExist"). This approach is based on storing very detailed traceability information about rule execution in order to determine when and how a rule should be re-executed for fixing inconsistencies; in contrast, the CDT approach is focused on reducing the amount of necessary information that is explicitly persisted in models.

[Tra08] presents a unidirectional, target incremental batch transformation language for model synchronization. Between two synchronization runs, the user may modify the source as well as target models, and the system will then propagate the changes incrementally, leaving manual target modi-
7.7. RELATED WORK

Specifications intact. This technique again relies on massive amounts of information explicitly cached in traceability models, by copying certain parts of the source model into traceability models.

Software evolution approaches

Software evolution approaches, which focus on the temporal development of system (meta)models, can be considered as a possible application area of the CDT technique, which could generate deltas (for different modeling domains) as inputs for the merging process required in software evolution. However, note that the CDT approach does not further support the merge conflict resolution subtask in any sense.

[Men02] lays down a wide-range terminology used in software evolution. According to this framework, snapshot, command, and history scenarios of Section 7.1 directly correspond to state-based, forward and backward delta approaches, respectively. Moreover, the CDT solution can be categorized as an operation and intensional change-based approach as model changes are explicitly expressed as transformations, and they are independent from the versions to which they are applied.

The FAMOOS project [DD99] whose aim was to build a framework to support the evolution and reengineering of object-oriented software systems used languages FAMIX [TDD00] and Hismo [GD06] for modeling purposes. More specifically, FAMIX is a language-independent model of object-oriented systems, which can be used for exchanging information between reengineering tools. FAMIX can be considered as a simplified metamodel for class diagrams without any support for describing changes. Hismo [GD06] extends metamodels by adding a time layer on top of the structural information, and it provides a common infrastructure for expressing and combining evolution and structural analyses. The additional time layer enables Hismo to support version control and to calculate changes of models, and in this sense, it could serve as a source of input for our approach, but Hismo has no metamodel for describing changes on a high abstraction level.

Visualization tools in the FAMOOS framework use side-effect-free OCL-based queries, which can even involve constructs from the time layer, but these queries are imperative from the viewpoint of structural constraint navigation, and they have been used for quantitative structural measurements (e.g., for counting the number of changed methods), in contrast to the CDT approach, which provides declarative graph patterns, which are used to drive and initiate the transformation of change descriptions. Additionally, the Goose tool in FAMOOS uses Prolog rules to search for violations of certain design guidelines. Prolog rules show similarity to graph patterns in their structure, however, the CDT approach requires no conversion of underlying models, in contrast to Goose, which can operate only on Prolog facts that have to be extracted in advance from FAMIX models.

[NLBK09] applies graph transformation for metamodel evolution in domain-specific languages. In this approach, GT rules evolve models in a metamodel compliance preserving way. More specifically, they describe the changes themselves inside a single modeling domain, but not the transformation of changes between different domains as in the CDT solution. Moreover, [NLBK09] lacks live transformation support.

Calculation of model correspondence and differences

Frameworks such as AMW [FB] allow discovering and representing hierarchical correspondences and differences between models. The approach presented by [SNG09] operates on a hierarchical traceability model to maintain high- and low-level correspondence between models, and outlines a mechanism for incrementally and efficiently maintaining traceability relationships. This technology
can also be used to create transformations that incrementally propagate changes to target models. The key challenge of these approaches is establishing this correspondence, using heuristics if necessary. Calculating differences (deltas) of models has been widely studied due to its important role in the process of model editing, which requires undo and redo operations to be supported. In [AP03], metamodel-independent algorithms are proposed for calculating directed (backward and forward) deltas, which can later be merged with the initial model to produce the resulting model. Unfortunately, the algorithms proposed by [AP03] for difference and merge calculation may only operate on a single model, and they are not specified by model transformation. In [CDRP07], a metamodel-independent approach is presented for visualizing backward and forward directed deltas between consecutive versions of models. Differences (i.e., change history models) have a model-based representation (similarly to [GKP07]), and calculations are driven by (higher order) transformations in both [CDRP07] and the proposed CDT approach. However, in contrast to [CDRP07] and [GKP07], CDT is applicable in a wide range of change scenarios (see Section 7.1).

Incremental synchronization for exogeneous model transformations

Various advantages of specifying transformations in terms of the deltas of models has been presented before in both [RVV09] (coining the term “change-driven” transformations) and [DXC10] (using the terminology “delta-based”). However, neither of these approaches propose a full CDT language that meets the challenges of Section 7.1.2.1.

Incremental synchronization techniques already exist in model-to-model transformation context (e.g., [XLH+07]). One representative approach is to use triple graph grammars [Sch95] for maintaining the consistency of source and target models in a rule-based manner. The proposal of [GW06] relies on various heuristics of the correspondence structure. Dependencies between correspondence nodes are stored explicitly, which drives the incremental engine to undo an applied transformation rule in case of inconsistencies. There are other triple graph grammar approaches for model synchronization (e.g., [KKS07]) that do not address incrementality. Triple graph grammar techniques are also used in [SMB05] for tool integration based on UML models. The aim of the approach is to provide support for change synchronization between various languages in several development phases. Based on an integration algorithm, the system merges changed models on user request. In this sense, contrarily to CDT, none of these approaches performs live transformation, but such a technique could possibly be easily integrated into these tools as well.

The approach of [Jim05] shows the largest similarity to CDT as both (i) focus on change propagation in the context of model-to-model transformation, (ii) describe changes in a metamodel-independent way, and (iii) use rule-driven algorithms for propagating changes of source models to the target side. In the proposal of [Jim05], the target model is finalized in a complex manual merge operation. In contrast, models using CDT are computed automatically on the target side.

7.8 Chapter conclusions

Change-driven transformations are a novel class of model transformations that are specified on the level of processing changes as their input. The chapter presented a novel language for specifying change-driven transformations, extending the well-established graph transformation language. I also outlined how the same language can be executed in different change scenarios by adapting incremental graph pattern matching engines. The language was successfully applied on issues raised in Chapter 6.


8

Conclusions

8.1 New scientific results

I summarize the novel scientific contributions of my PhD thesis below. Each of the contributions stated here corresponds respectively to one of the goals that were set in Section 1.2.3.

8.1.1 Efficient, incremental pattern matching in a model-driven environment

In case of large-scale models, execution time may be a critical factor in the success of model transformation. A possible way to speed up transformations and queries is the application of incremental techniques. Source incrementality is not unavailable in most model transformation frameworks, and incremental graph pattern matching based approaches were in an early phase [BGT91, VVS06] of investigation at the beginning of my research.

Contribution 1

I have adapted incremental algorithms of expert systems to realize graph pattern matching over large evolving models. I have demonstrated the efficiency of the approach in different application scenarios with performance measurements.

1. Adapting Rete for incremental graph pattern matching. I introduced a general theoretical framework for the semantics and algorithmic complexity of incremental pattern matching. In this context, I formalized the Rete [For82] algorithm from the field of rule-based expert systems. I have implemented the algorithm to operate on a rich graph pattern language as used in a model-driven context. [23]

2. Parallel incremental pattern matching. I proposed parallel execution methods for incremental model query evaluation. I have identified three ways of parallelization: (a) concurrent execution of model manipulation and pattern matching (the maintenance phase in particular), and applying a multi-threaded strategy separately and independently (b) in model manipulation and (c) in matcher maintenance. [26,4]

3. Extending incremental pattern matching by transitive closure. I proposed an efficient, incremental query evaluation method for handling generic transitive closure of graph edges and binary graph patterns. [8]
4. **Adapting incremental graph pattern matching to relational databases.** I proposed an incremental method of query evaluation over models persisted in relational databases, which integrates with existing data manipulation software. I proposed a mapping from graph patterns to an event-driven SQL program that implements the TREAT [ML91] incremental algorithm. [7]

5. **Quantitative performance analysis of incremental graph pattern matching.** I demonstrated the efficiency of the proposed incremental pattern matching strategy on model transformation benchmarks. I have identified scenarios (such as behavioral model simulation, M2M live synchronization) when its application is beneficial compared to traditional search-based approaches [16,4,14,2]

Several research results of my colleagues István Ráth and Ákos Horváth rely on the incremental pattern matcher I have implemented in the Vi.t/a.e.sc/a.t.sc/r.a.sc2 framework, including simulation-based analysis of DSM languages [RVV08], model-based design constraint satisfaction by design space exploration [HV09, HV10], or stochastic graph transformation [3][THR10, KHTR10]. I applied the incremental query technique for simulation-based calibration of sensor networks [13].

The core idea of incremental pattern matching was presented in the DSc thesis of my PhD supervisor Dániel Varró. I have developed the prototype implementation used for my investigations as a pattern matcher plug-in module for the Vi.t/a.e.2 [VIA] model transformation framework. Gergely Varró offered extensive help in starting my research as my master thesis supervisor. Elaborating measurements was joint work with István Ráth and Ákos Horváth, and the strategies for combining different pattern matching approaches is now part of the PhD thesis of Ákos Horváth. Under my supervision, Tamás Szabó contributed the prototype implementation and conducted the measurements of incremental pattern matching extended by generic transitive closure. Under my supervision, Dóra Horváth contributed a prototype implementation and performed the measurements for incremental pattern matching over relational databases.

### 8.1.2 Incremental model queries over industrial EMF models

Model queries have various use cases in MDE. My aim is to allow declarative specification of these queries in a high-level language, and enable efficient evaluation.

Industrially accepted technologies offer various languages for specifying model queries (such as OCL [OMG12a]), but these formalism cannot always easily express the connection structure of several objects, which is important for use cases such as complex well-formedness constraints. Furthermore, most query evaluators for these languages are not incremental, and the exceptions are mostly academic tools. Therefore industrial platforms such as EMF could benefit greatly from a graph pattern based language that would be able to express complex queries that would be efficiently evaluated by an incremental matcher.

**Contribution 2** I proposed a declarative and expressive query language for specifying queries over the industrial Eclipse Modeling Framework. I have designed an incremental pattern matcher for the efficient evaluation of these queries. I have demonstrated the efficiency of the approach by performance measurements.
1. **Graph pattern based query language for EMF models.** I proposed a graph pattern based model query language for EMF models. The syntax is based on the pattern language [BV06] of Viatra2, extending it by new features including path expressions, and adjusting it to the characteristics of the EMF model representation. [10]

2. **Incremental evaluation for EMF model queries.** I integrated the Rete-based incremental pattern matcher algorithm into the context of EMF models, and provided a translation that performs Rete maintenance according to the EMF notification scheme [11,9,28,29,24].

3. **Performance analysis of model query frameworks over EMF.** I have demonstrated the expressiveness of the proposed language and the efficiency of incremental evaluation based on static model validation problems from the automotive domain. [11]

The EMF-IncQuery model query technology has been presented at multiple public tutorial sessions (including [28,29]), and gained significant attention in both academic and industrial audiences. These presentations have been a joint work with István Ráth, Ákos Horváth, Ábel Hegedüs, and others.

The defined query language contains ideas of István Ráth, Zoltán Ujhelyi and the authors of the original VTCL pattern syntax [BV06] on which it is based, and has since been used in further research (e.g. [17]). I built the prototype implementation into the EMF-IncQuery [11] tool in cooperation with Ákos Horváth. The above tutorials were joint work of this team. The case study and experiments from the domain of automotive industry is joint work with our industrial co-authors (András Balogh, Zoltán Balogh, András Ökrös). As a result of our joint work, EMF-IncQuery is now an official part of the Modeling project of the Eclipse Foundation [ECLb].

### 8.1.3 Supporting change-driven transformation specification by queries

The specification of transformations which process evolving models can greatly benefit from change-driven reactions. While some modeling platforms (VPM, EMF) provide notifications of elementary model deltas, their granularity is too low. In order to support the detection of complex changes, all preceding changes and their context must be taken into account. My results provide these capabilities for the concept of change-driven transformations that was proposed in [RVV09].

A further challenge is posed by the wide range of application scenarios where change detection may be necessary. One of these cases is live transformation (see [15]), where a continuously active transformation reacts immediately to model changes. However, if live transformation is not applicable, processing changes requires a different strategy.

**Contribution 3** By extending the formalism of graph patterns, I designed a new change pattern language for high-level, context-aware detection of structural changes of models. For each application scenario characterized by the information available on the model and its changes, I have proposed a dedicated strategy to efficiently evaluate a change pattern according to its formal semantics.
1. **Categorizing change scenarios.** I proposed a taxonomy of application scenarios for change-driven transformations, based on the available information describing the model and its change (such as model difference, change notification, archive version). [1]

2. **Language for defining change patterns.** I designed a change-driven transformation language based on graph patterns. The language can express queries against the changes of a model, independently of the application scenario. [25,27,1]

3. **Formal semantics of change patterns.** I formally defined the match set of a change pattern, in context of the change of a model. [1]

4. **Pattern matcher strategies for change-driven transformations** I have designed strategies for evaluating change patterns in accordance with their formal semantics. There is a separate implementation strategy for each of the identified change scenarios, which efficiently computes the match set of change patterns based on the information available in the specific change scenario. [15,1]

The proposed results were used by Ábel Hegedüs for back-propagating simulation results [12]. My work extends the concepts of change-driven transformations [RVV09], used in the PhD thesis of István Ráth.

### 8.1.4 Queries and transformation in modeling security requirements

In requirement modeling, requirements may have to be represented in multiple formalisms. Moreover, in security engineering processes, the requirement model is often interrelated with other models. Security experts can investigate security issues in requirements in a long and costly process, and their system of arguments can only be recorded in structured informal models. Automatic detection of simple security problems is not supported, and if the requirements evolve, all arguments need to go through the costly process of re-evaluation.

#### Contribution 4

I designed an integrated environment for security requirement analysis, by using model queries and change-driven transformations,

1. **Bidirectional change-driven synchronization between security requirement models** I proposed change-driven, live transformations to support security requirement elicitation and analysis. I designed an environment architecture with a central abstract model (conforming to the SeCMER [MMP+11] formalism) in a bidirectional synchronization relationship with a different requirement model syntax (Si* [MMZ07]). [19]

2. **Continuous validation of security criteria over evolving requirement models.** I proposed automated analysis of security requirements to check simple security criteria and identify violations. I applied graph pattern based queries to formalize the security constraints. I designed an implementation architecture where the requirements engineer is continuously informed by problem markers maintained according to incrementally evaluated queries. [19]
8.2 Future directions

There are a number of areas where I envision significant future improvements to current results.

The Rete algorithm has been chosen as the incremental pattern matching engine behind most of the scientific contributions of this thesis. It would be interesting to adapt other incremental matcher algorithms that have likewise emerged from the field of rule-based expert systems (see Section 3.2.3 on page 44), and contrast their performance characteristics against that of Rete in the use cases specific to model-driven engineering (as opposed to earlier comparisons designed for expert systems). Note that one of my experiments (see Section 4.3 on page 87) has already been performed with a matcher based on TREAT [ML91].

Remaining in the context of Rete, parallelization offers a different path towards increasing performance. There are three approaches introduced in Section 4.1 on page 73, two of which can demonstratively speed up execution under the right circumstances. The third approach, which distributes a Rete net into several containers (execution threads), still lacks good heuristics before its potential can be realized. Discovering such efficient distribution strategies is left for future research.

A final way to improve performance is to apply various query plan optimization strategies and other improvements to the Rete implementation; these will also be investigated in the future.

An important limitation of the pattern matching strategy used throughout the thesis is that it does not fully support recursive pattern composition, unlike some other approaches [VHV08, HJG08]. There exists at least one approach [GMS93] that is both incremental and capable of dealing with recursively formulated queries; investigating the applicability and performance of such algorithms is an important future work. Note though that my proposed incremental strategy includes a solution for a special case of recursive pattern composition, namely transitive closure computation (see Section 4.2 on page 80). Our implementation has been shown to perform well, although comparisons against alternative approaches have not yet been performed.

In Chapter 7 on page 129, I have proposed a high-level query and rule formalism for change-driven transformations. The newly introduced change-driven graph transformation systems currently lack such analysis techniques that are already known for graph transformation; adapting or inventing suitable methods is a significant research direction to consider into the future. Additionally, I plan to investigate how change-driven graph transformation systems can simulate well-established model transformation specification formalisms, in order to grant them change-driven execution properties.
8.3 Applications of new scientific results

Finally, I showcase some practical applications of my new conceptual results.

8.3.1 Incremental pattern matcher module of the VIATRA2 model transformation framework

VIATRA2 is a general-purpose graph transformation-based model transformation framework, which is part of the Generative Modeling Technologies project [ECLA] of the Eclipse Foundation [ECLb]. It has been developed for almost 10 years at the Department of Measurement and Information Systems, Budapest University of Technology and Economics. The incremental graph pattern matching module of the current VIATRA2 version is built on conceptual results of Contribution 1.

VIATRA2 itself has been applied in numerous international research projects, for tool integration (DECOS FP6, DIANA FP6 [6], MOGENTES FP7, SecureChange FP7 [EU 12] EU projects), model validation (HIDENETS FP6 EU project), source code synthesis (SENSORIA FP6 [5], E-Freight FP7 EU projects), and even behavior model simulation (sensor network analysis [13] in French-Hungarian intergovernmental project). VIATRA2 has regularly appeared in tool contests for transformation frameworks [34,35], where the incremental pattern matcher module was used.

8.3.2 EMF-InCQuery

A recent project of the developer group behind VIATRA2 is the EMF-InCQuery [11] framework, which enables wide-spread immediate application of results in VIATRA2 and Contribution 2 on the EMF platform. The main run-time component of EMF-InCQuery implements the EMF-based query language and incremental pattern matching method of Contribution 2.

Through EMF-InCQuery, many of the results of the thesis can now be integrated with numerous open and proprietary products. Our research group, partners and early external adopters have already applied the tool in several projects. The tool was used in multiple national research grants (Jedlik, CertiMoT, TÁMOP) and for the realization of the SeCMER tool prototype (see Section 8.3.3) in the EU FP7 project SecureChange.

EMF-InCQuery already has a number of foreign uses. At least the following organizations have introduced EMF-InCQuery to their development practice, or conducted pilot investigations:

- Thales Group
- Itemis AG
- Obeo
- ThyssenKrupp Presta Hungary Ltd
- Montages
- evopro Informatikai és Automatizálási Kft.
- CERN
- CEA
- INRIA
8.3. APPLICATIONS OF NEW SCIENTIFIC RESULTS

- TU München
- KU Leuven
- University of York
- University of Nantes
- Austria Institute of Technology
- TU Eindhoven
- Universität Innsbruck

The following example applications of EMF-IncQUERY have all been carried out independently of me:

- Incremental dependency analysis in large source code models at CERN
- Detection of change patterns in security architecture modeling at KU Leuven
- Declarative definition and incremental maintenance of derived features at Itemis
- Driving test oracles in MT testing at University of Nantes
- Providing query-driven soft interconnection of EMF models at BME

8.3.3 SeCMER tool prototype

The EU FP7 project SecureChange [EU 12] is concerned with the evolution of security critical systems; the demonstrator tool [19] of the project for security requirements engineering relies on several of my results. The tool provides the query and transformation based support proposed in Contribution 4, and EMF-IncQUERY (see Section 8.3.2) played a big role in its implementation.

The validation of this demonstrator tool was performed in September 2011, according to the rules of the SecureChange project. At the validation event, the tool was presented through an air traffic management case study to participating flight security and air traffic control experts, who provided feedback that was incorporated in the tool.

The SeCMER prototype was highly acknowledged by project reviews.
List of publications

Number of publications: 36
Number of peer-reviewed publications: 26
Number of known independent citations: over 120

Journal papers (4)


Book chapters (2)


International conferences (10)


**International workshops (7)**


**Domestic conference proceedings (3)**

CHAPTER 8. CONCLUSIONS


Hungarian-language conference article (1)

Tutorials (3)


EU research project deliverables (3)


[33] Michela Angeli, Karmel Bekoutou, Gábor Bergmann, Elisa Chiarani, Olivier Delande, Edith Felix, Fabio Massacci, Bashar Nuseibeh, Federica Paci, Thein Tun, Dániel Varró, Koen

**Reports (2)**


**Master’s thesis (1)**

Bibliography


